الجامعة السعودية الإلكترونية

SAUDI ELECTRONIC UNIVERSITY

2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 1
Introduction

Contents

1. Saudi Vision 2030
2. Saudi Digital Library
3. Git and Github
4. Stackoverflow

Weekly Learning Outcomes

- Demonstrate the Saudi Digital Library and searching for knowledge
- Understand the Major Objective of Saudi Vision 2030

- Saudi Vision 2030

Saudi Vision 2030

- Under the leadership of the Custodian of the Two Holy Mosques, Vision 2030 was launched, a roadmap drawn up by His Royal Highness the Crown Prince, to harness the strengths God bestowed upon us – our strategic position, investment power and place at the center of Arab and Islamic worlds. The full attention of the Kingdom, and our Leadership, is on harnessing our potential to achieve our ambitions.

Reference: https://www.vision2030.gov.sa/

Saudi Vision 2030

- **Vision 2030 Draws on The Nation’s Intrinsic Strengths**

  1. Saudi Arabia is the land of the Two Holy Mosques which positions the Kingdom at the heart of the Arab and Islamic worlds

  2. Saudi Arabia is using its investment power to create a more diverse and sustainable economy

  3. The Kingdom is using its strategic location to build its role as an integral driver of international trade and to connect three continents: Africa, Asia and Europe

Reference: https://www.vision2030.gov.sa/

- Saudi Digital Library

Saudi Digital Library

- Saudi Digital Library, is the largest academic gathering of information sources in the Arab world, with more than (310،000) scientific reference, covering all academic disciplines, and the continuous updating of the content in this; thus achieving huge accumulation cognitive in the long run. Library has contracted with more than 300 global publisher. The library won the award for the Arab Federation for Libraries and Information ‘know’ for outstanding projects in the Arab world in 2010.

Reference: https://sdl.edu.sa/sdlportal/en/publishers.aspx

Saudi Digital Library

- Advantages:
  - One central management, manages this huge content, and constantly updated.
  - Common share for the benefit of, any University would benefit other universities that are now available to the other, in any scientific field.
  - Enhance the status of universities when evaluating, for Academic Accreditation, and through sources rich, modern, and publish the best Global Publishers.
  - Bridging the gap between Saudi universities, where emerging universities can get the same service, you get major Saudi universities.

Reference: https://sdl.edu.sa/sdlportal/en/publishers.aspx

- Git and Github

What is GIT?

- **Git**
  - is a version control system for tracking changes in computer files and coordinating work on those files among multiple people.
  - It is primarily used for source code management in software development and was initially created by Linus Torvalds for development of the Linux Kernel.

- **Git** is not **Github**.
  - **Git** is the version control software
  - **Github** is a git repository hosting service which offers all the source code management provided in git.
  - **Github** is where you upload your git repository.

Reference: https://medium.com/@lucasmaurer/git-gud-the-working-tree-staging-area-and-local-repo-a1f0f4822018

Version control

- **Version control** is:
  - a system that records changes to a file or set of files over time so that you can recall specific versions later.
  - If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use.
  - It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.
  - Using a VCS also generally means that if you lose files, you can easily recover.

Reference: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

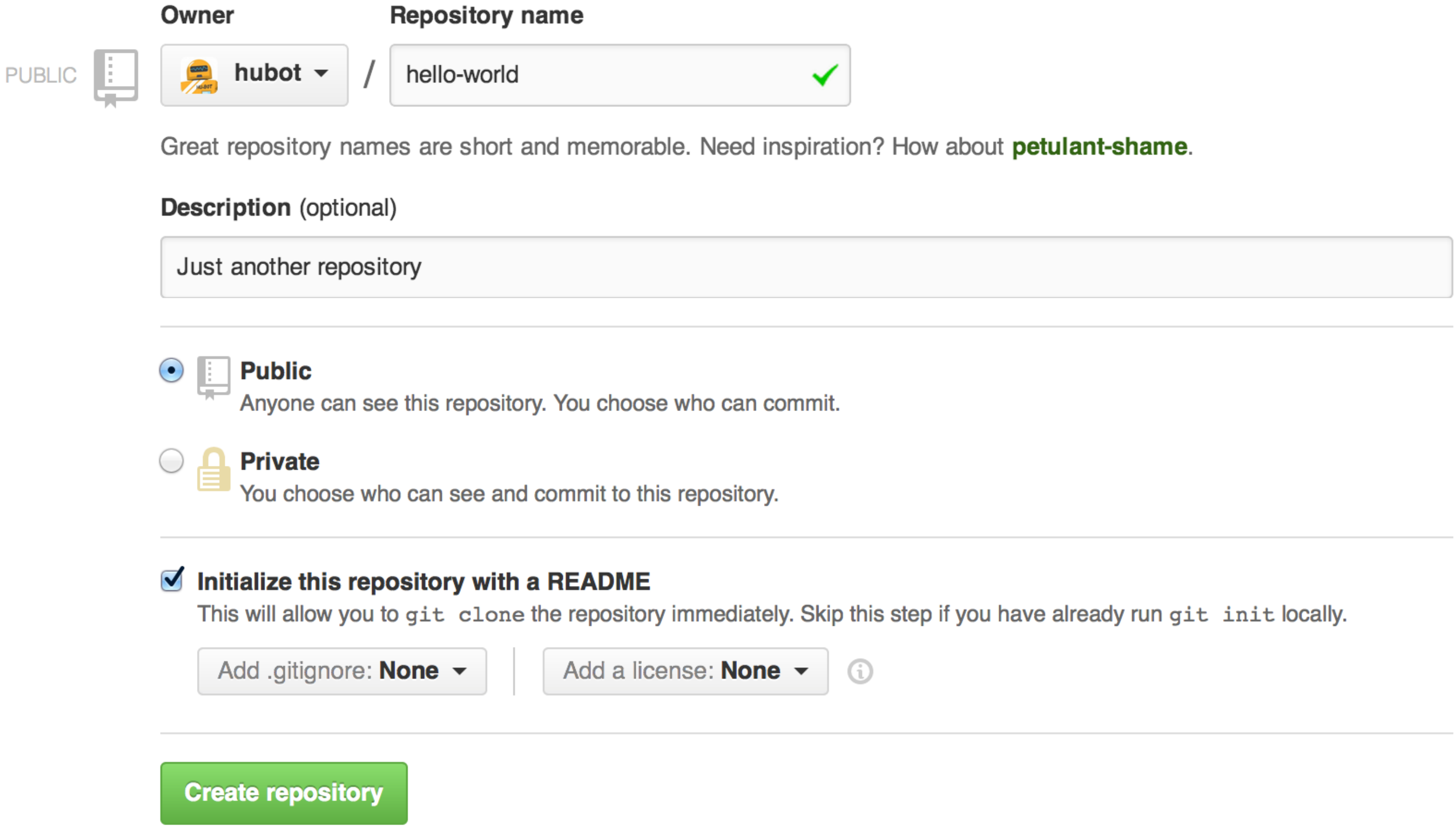
What Is GitHub

- **GitHub** is
  - a for-profit company that offers a cloud-based **Git** repository hosting service.
  - Essentially, it makes it a lot easier for individuals and teams to use **Git** for version control and collaboration.

- **GitHub's** interface is user-friendly enough so even novice coders can take advantage of Git.

Reference: https://guides.github.com/activities/hello-world/

Create a Repository



PUBLIC

Owner

hubot

/

Repository name

hello-world ✓

Great repository names are short and memorable. Need inspiration? How about **petulant-shame**.

Description (optional)

Just another repository

● **Public**
Anyone can see this repository. You choose who can commit.

○ **Private**
You choose who can see and commit to this repository.

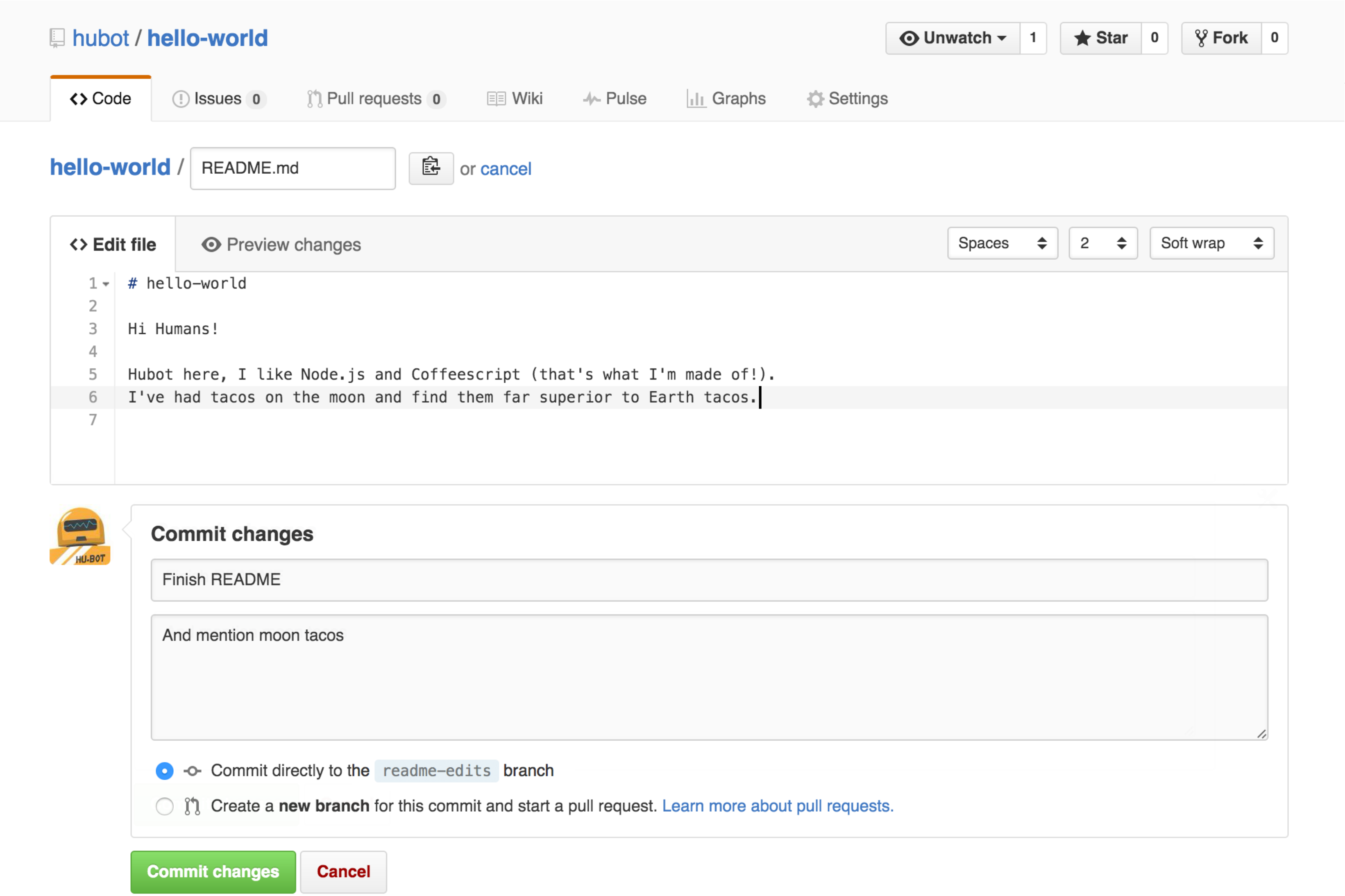
☑ **Initialize this repository with a README**
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None**

Add a license: **None**

Reference: https://guides.github.com/activities/hello-world/

Make and commit changes



```
# hello-world

Hi Humans!

Hubot here, I like Node.js and Coffeescript (that's what I'm made of!).
I've had tacos on the moon and find them far superior to Earth tacos.

```

Commit changes

Finish README

And mention moon tacos

● Commit directly to the `readme-edits` branch

○ Create a **new branch** for this commit and start a pull request. Learn more about pull requests.

Reference: https://guides.github.com/activities/hello-world/

- **Stack overflow**

Stack overflow

- **Stack Overflow** is
  - a question-and-answer website for professional and enthusiast programmers.
  - It is the flagship site of the Stack Exchange Network, created in 2008 by Jeff Atwood and Joel Spolsky.
    - It features questions and answers on a wide range of topics in computer programming.
    - Stack Overflow only accepts questions about programming that are tightly focused on a specific problem.
    - Questions of a broader nature–or those inviting answers that are inherently a matter of opinion– are usually rejected by the site's users and marked as closed.

https://stackoverflow.com/

Reference: https://en.wikipedia.org/wiki/Stack_Overflow

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 2

Introduction to Computer Science and Programming

Contents

1. Introduction to Hardware and Software
2. Low-Level and High-Level Programming Language
3. Operating System
4. Java Development Environment
5. Internet and World Wide Web

Weekly Learning Outcomes

- Illustrate Low-Level vs High-Level Programming Languages
- Recognize the difference between Compiler and Interpreter

Required Reading

1. Chapter 1 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 1 - (Java The Complete Reference, Eleventh Edition)
2. Difference between Compiler and Interpreter:

   https://www.javatpoint.com/compiler-vs-interpreter
3. NetBeans Tutorials: https://netbeans.apache.org//kb/index.html

- Introduction to Hardware and Software

Introduction

- Java one of the world’s most widely used computer programming languages. You’re already familiar with the powerful tasks' computers perform.
- For many organizations, the preferred language for meeting their enterprise programming needs is Java.

- Java Editions:
  - Java Standard Edition
    - Contains the capabilities needed to develop desktop and server applications.
  - Java Enterprise Edition
    - Is geared toward developing large-scale, distributed networking applications and web-based applications

Hardware and Software

- Many of today’s personal computers can perform billions of calculations in one second
  - A computer consists of various devices referred to as hardware
- Computers process data under the control of sequences of instructions called computer programs*.*
  - These software programs guide the computer through ordered actions specified by people called computer programmers*.*
- Hardware
  - Physical devices of computer system
- Software
  - Programs that run on computers

Hardware and Software

- **Computer Organization**
  - Six logical units of computer system
    - Input unit
      - Mouse, keyboard
    - Output unit
      - Printer, monitor, audio speakers
    - Memory unit
      - RAM
    - Arithmetic and logic unit (ALU)
      - Performs calculations
    - Central processing unit (CPU)
      - Supervises operation of other devices
    - Secondary storage unit
      - Hard drives, floppy drives

Hardware and Software

- **Computer Organization**

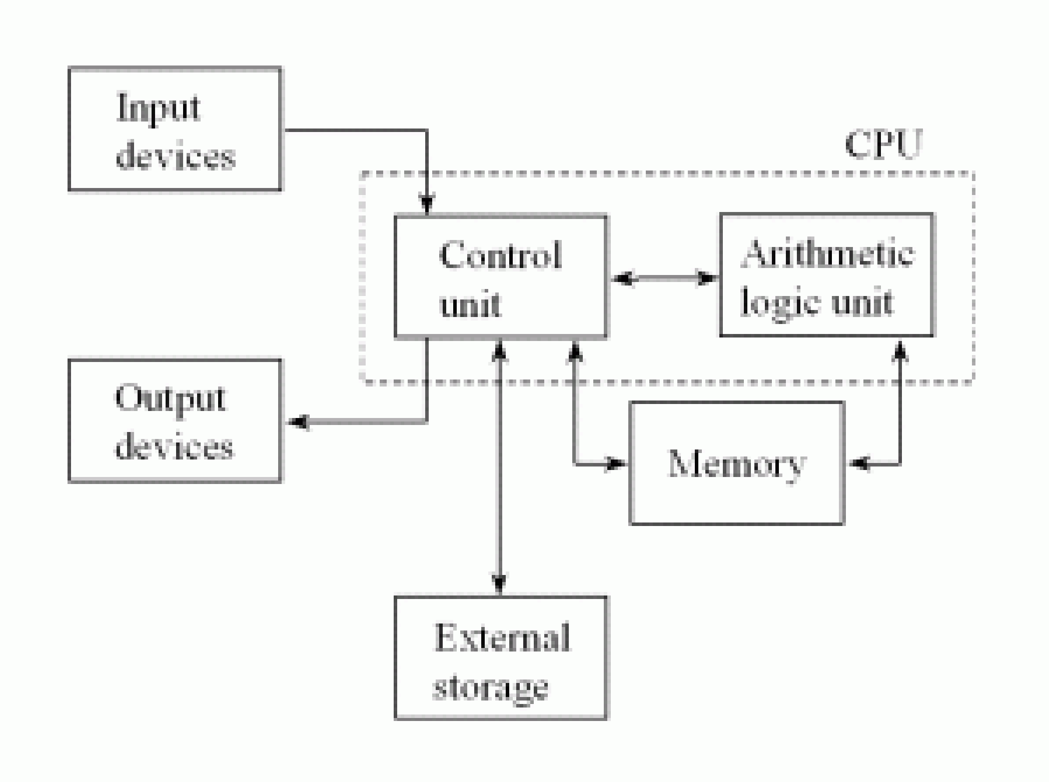
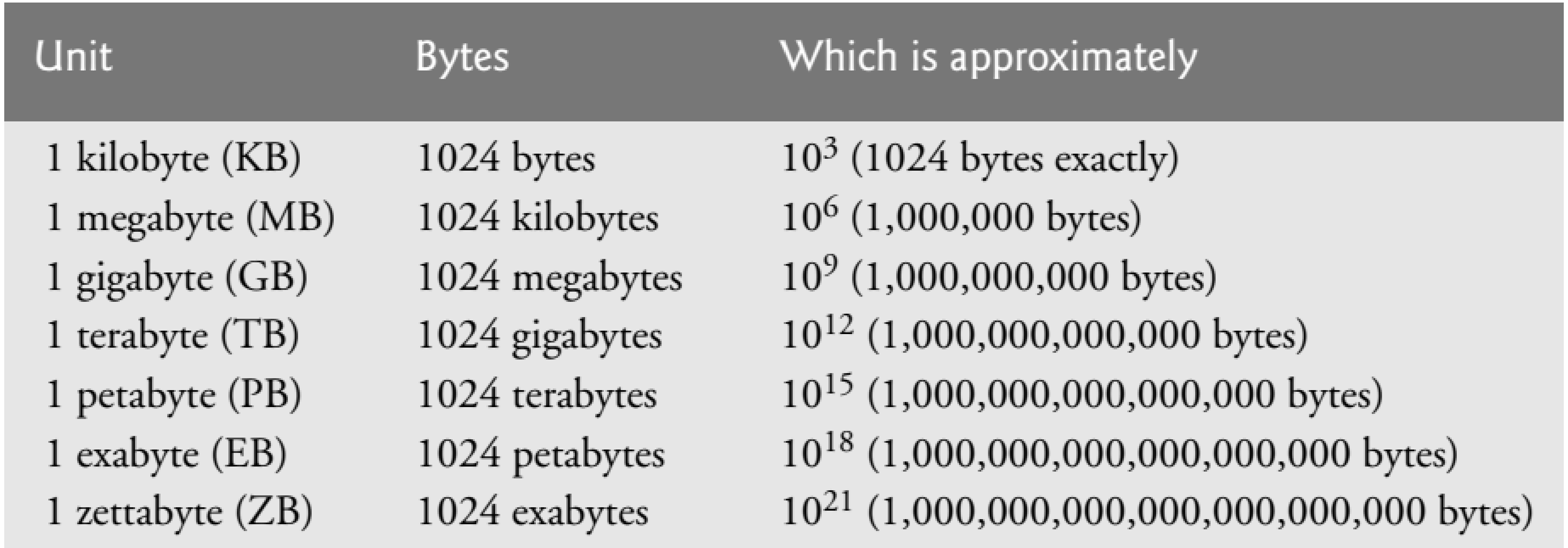


Figure: https://naoreenchowdhury.wordpress.com/computer-organization-architecture/

Hardware and Software

- **Data Hierarchy**
  - Bits
  - Characters
  - Fields
  - Records
  - Files
  - Database
  - Big Data



Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10^3 (1024 bytes exactly)
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000 bytes)
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000 bytes)
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000 bytes)
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000 bytes)
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000 bytes)
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000 bytes)

Figure: Chapter 1 - (Java: How to program: Late Objects)

- Low-Level and High-Level Programming Language

Machine Languages

Any computer can directly understand only its own machine language, defined by its hardware design.

Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time.

Machine languages are *machine dependent*

Assembly Languages and Assemblers

- Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations.

- These abbreviations formed the basis of assembly languages.

- *Translator programs* called assemblers were developed to convert early assembly-language programs to machine language at computer speeds.

High-Level Languages and Compilers

- To speed the programming process, high-level languages were developed in which single statements could be written to accomplish substantial tasks.
- Translator programs called compilers convert high-level language programs into machine language.
- High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations.

Interpreters

- Compiling a large high-level language program into machine language can take considerable computer time.

- *Interpreter* programs, developed to execute high-level language programs directly, avoid the delay of compilation, although they run slower than compiled programs.

- Operating System

Operating System

- It is software systems that make using computers more convenient for users, application developers and system administrators.

- They provide services that allow each application to execute safely, efficiently and *concurrently* (i.e., in parallel) with other applications.

- The software that contains the core components of the operating system is the kernel

Operating System

Windows

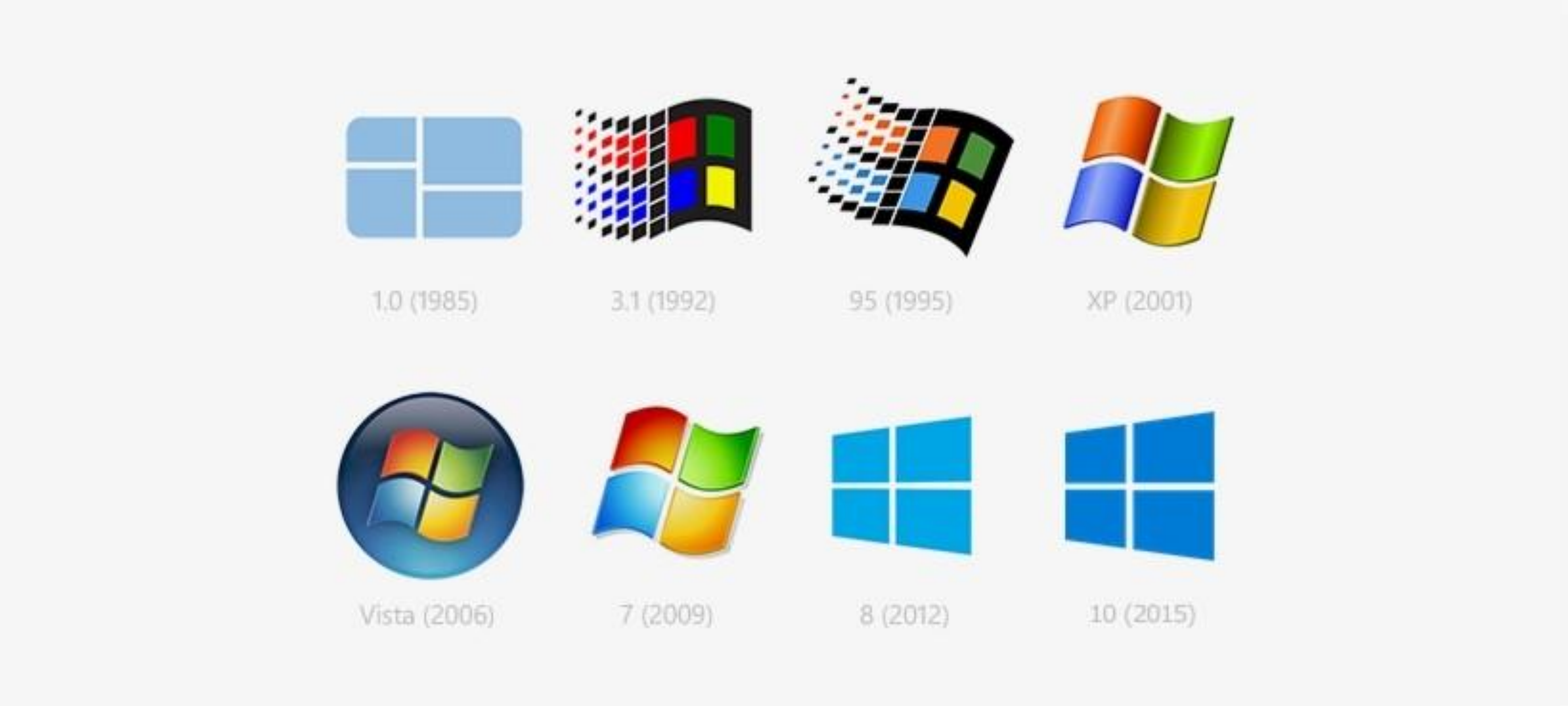


Figure: https://www.almrsal.com/post/955105

(2021)

Operating System

- **Linux**

Figure: https://analyticsindiamag.com/10-most-stable-linux-distros-in-2021/

Operating System

MAC

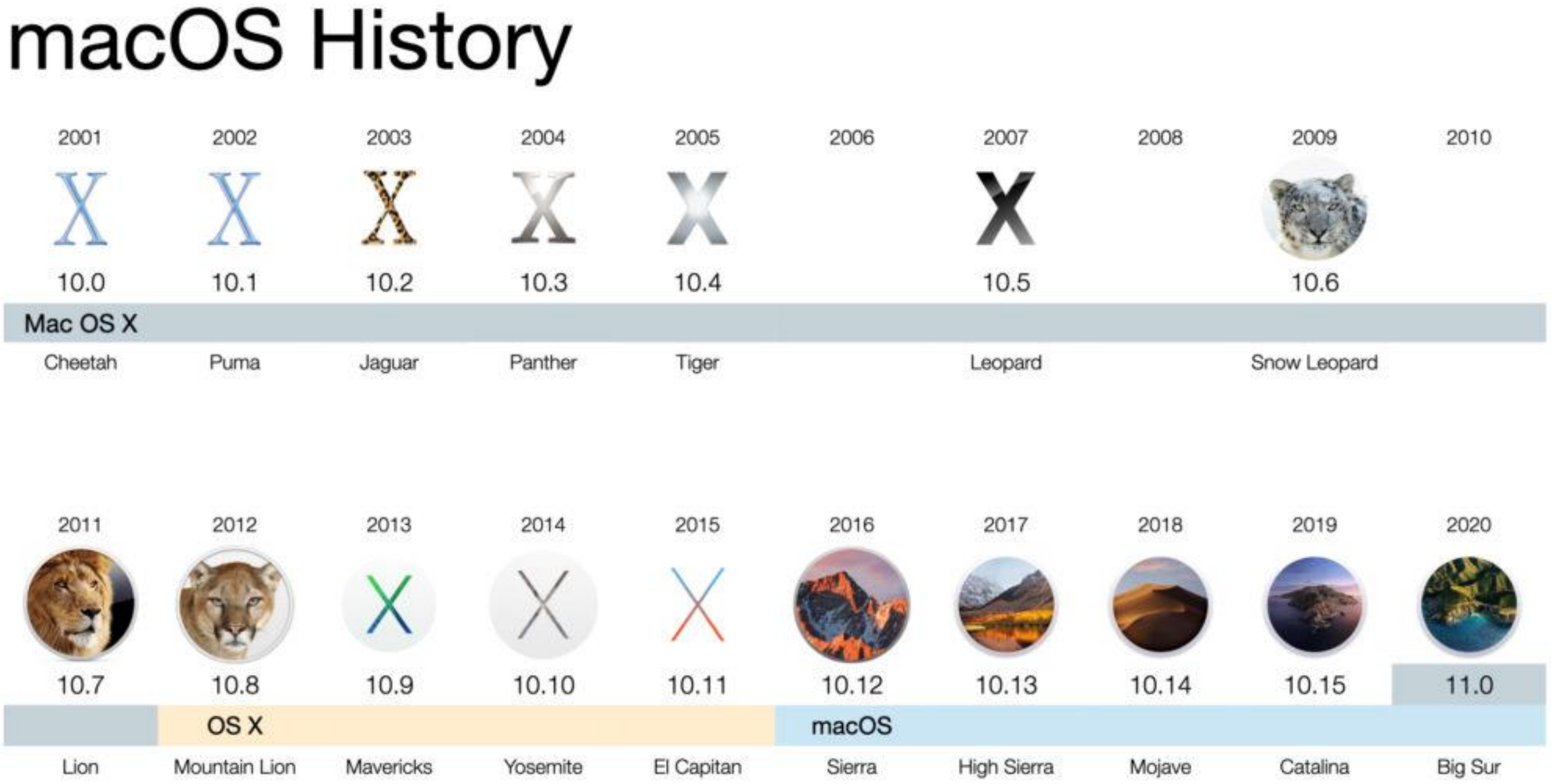


Figure: https://scriptingosx.com/2021/03/twenty-years-of-mac-os-x/

Operating System

Android

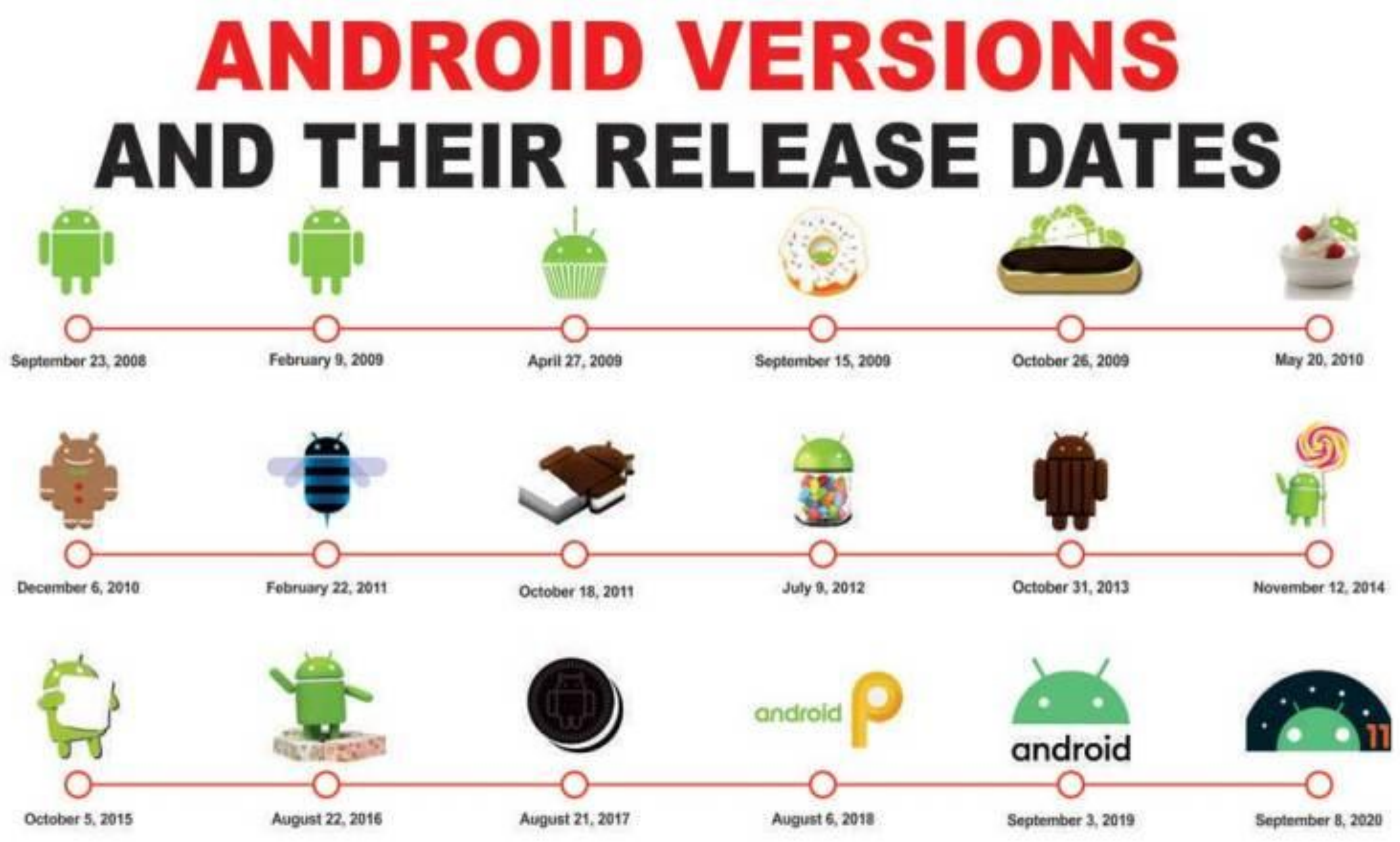


Figure: https://csquery.com/android-versions-and-their-release-dates/

- Programming Languages

Programming Languages

Figure: https://medium.com/analytics-vidhya/what-programming-language-should-you-learn-first-ff6e8f9c5000

Programming Languages

- **Java**
  - A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called “write once, run anywhere.”
  - Developed by Sun Microsystems in 1991. Sun Microsystems was acquired by Oracle in 2010.

Programming Languages

- **Java Class Libraries**
  - Classes
    - Contain *methods* that perform tasks
      - Return information after task completion
    - Used to build Java programs
  - Java contains class libraries
    - Known as Java APIs (Application Programming Interfaces)

- Java Development Environment

Java Development Environment

- *Phase 1: Creating a Program*
  - consists of editing a file with an *editor program*, normally known simply as an *editor*
  - Java source code files are given a name ending with the **.java** extension

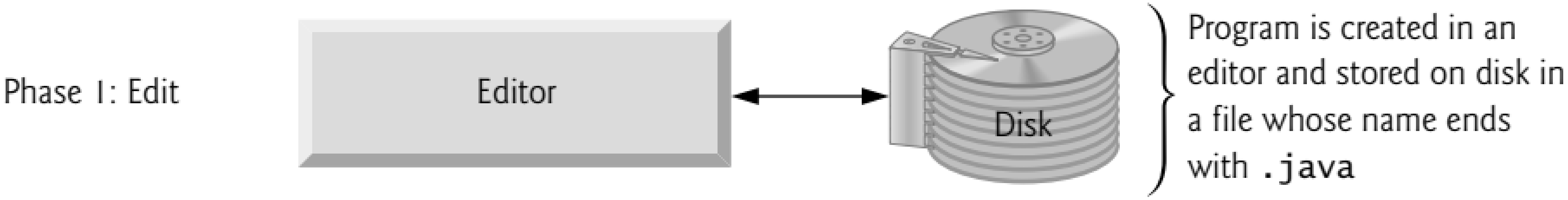

Phase 1: Edit
Editor
Disk
Program is created in an editor and stored on disk in a file whose name ends with .java

Figure 1.6: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- *Phase 1: Creating a Program*
  - Integrated development environments (IDEs) provide tools that support the software development process, such as editors, debuggers for locating logic errors.

    - Eclipse (www.eclipse.org)
    - NetBeans (www.netbeans.org)
    - IntelliJ IDEA (www.jetbrains.com)
    - BlueJ (www.bluej.org)

Java Development Environment

- *Phase 2: Compiling a Java Program into Bytecodes*
    - You use the command **javac** (the Java compiler) to compile a program
    - If the program compiles, the compiler produces a **.class** file called Welcome.class that contains the compiled version.
    - IDEs typically provide a menu item, such as Build or Make, that invokes the javac command for you.

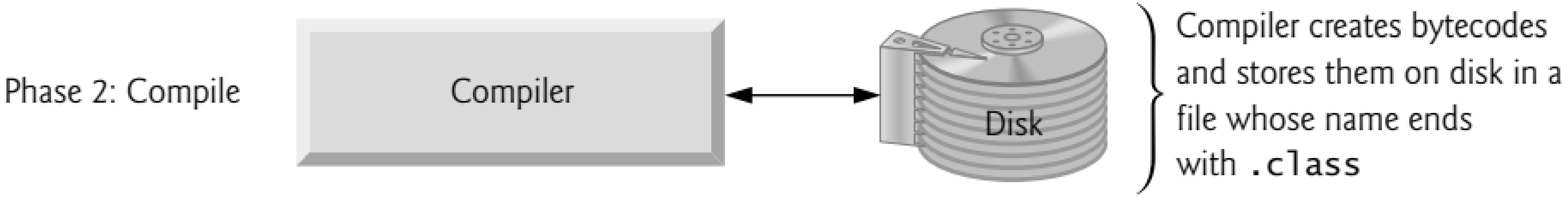

Phase 2: Compile
Compiler
Disk
Compiler creates bytecodes and stores them on disk in a file whose name ends with .class

Figure 1.7: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- *Phase 2: Compiling a Java Program into Bytecodes*
  - The Java compiler translates Java source code into bytecodes that represent the tasks to execute in the execution phase.
  - The Java Virtual Machine (JVM)—a part of the JDK and the foundation of the Java platform—executes bytecodes.
    - A virtual machine (VM) is a software application that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it.

machine-language instructions	bytecode instructions
platform dependent	*platform independent*
Non-portable	portable
You need to recompile the source code on new machine.	No need to recompile the source code on new machine

Java Development Environment

- *Phase 3: Loading a Program into Memory*
  - In this phase the JVM places the program in memory to execute it—this is known as loading .
  - The JVM's class loader takes the .class files containing the program's bytecodes and transfers them to primary memory.

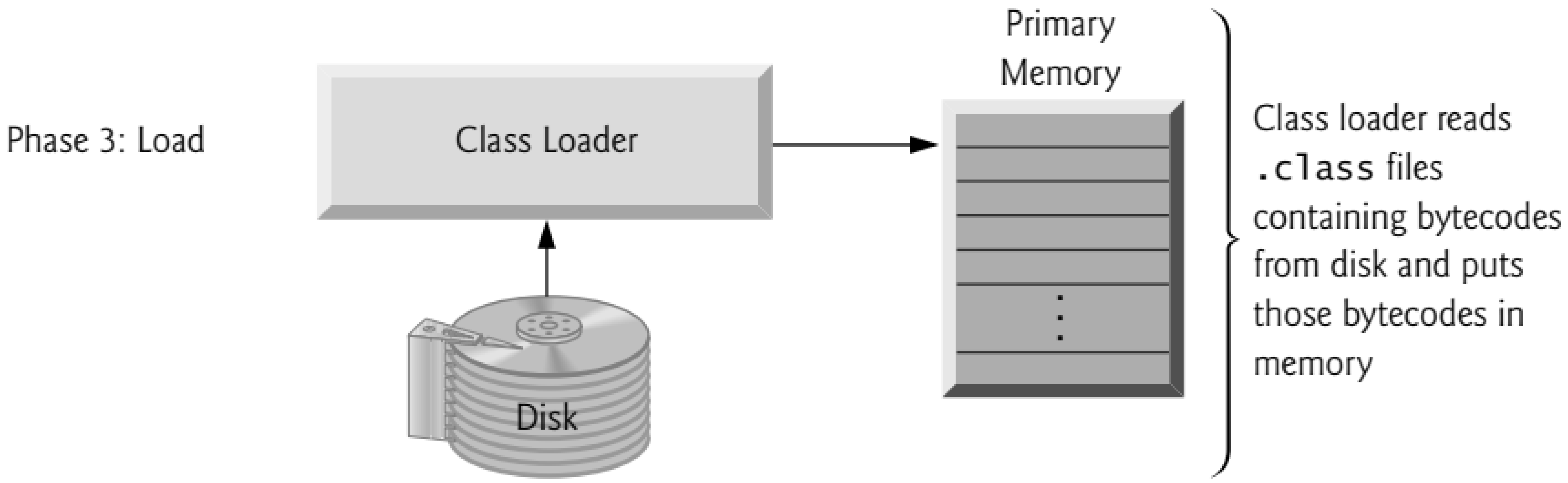

Phase 3: Load
Class Loader
Disk
Primary Memory
Class loader reads .class files containing bytecodes from disk and puts those bytecodes in memory

Figure 1.8: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- *Phase 4: Bytecode Verification*
  - After the classes are loaded, the bytecode verifier examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions.
  - Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system.

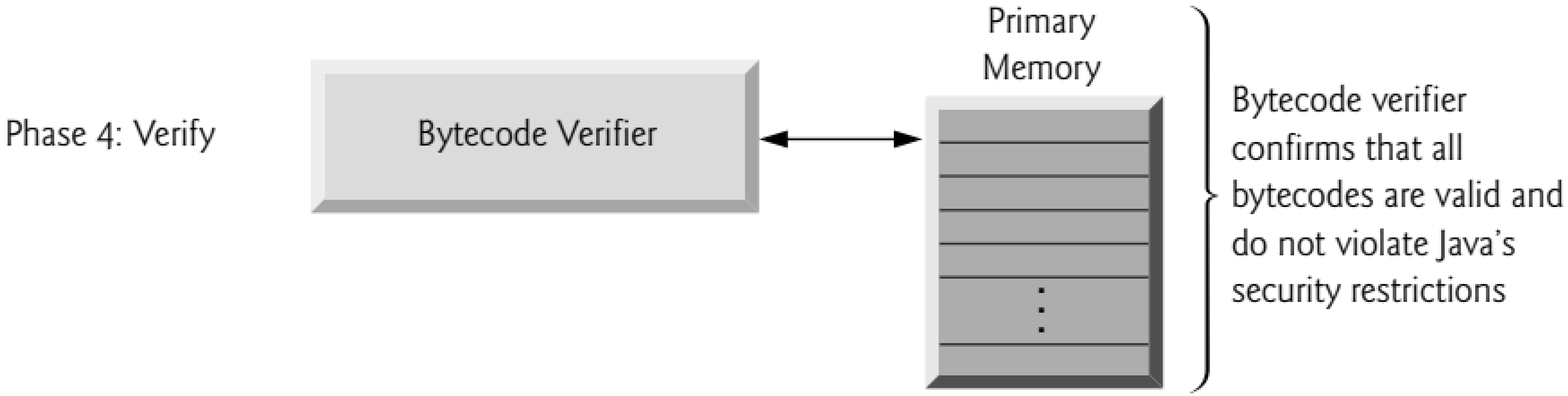

Phase 4: Verify
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions

Figure 1.9: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- *Phase 5: Execution*
  - The JVM executes the program's bytecodes, thus performing the actions specified by the program.
  - Today's JVMs typically execute bytecodes using a combination of interpretation and so-called just-in-time (JIT) compilation.
    - Searching for *hot spots*—parts of the bytecodes that execute frequently. For these parts, a just-in-time (JIT) compiler, translates the bytecodes into the underlying computer's machine language.
    - When the JVM encounters these compiled parts again, the faster machine-language code executes.

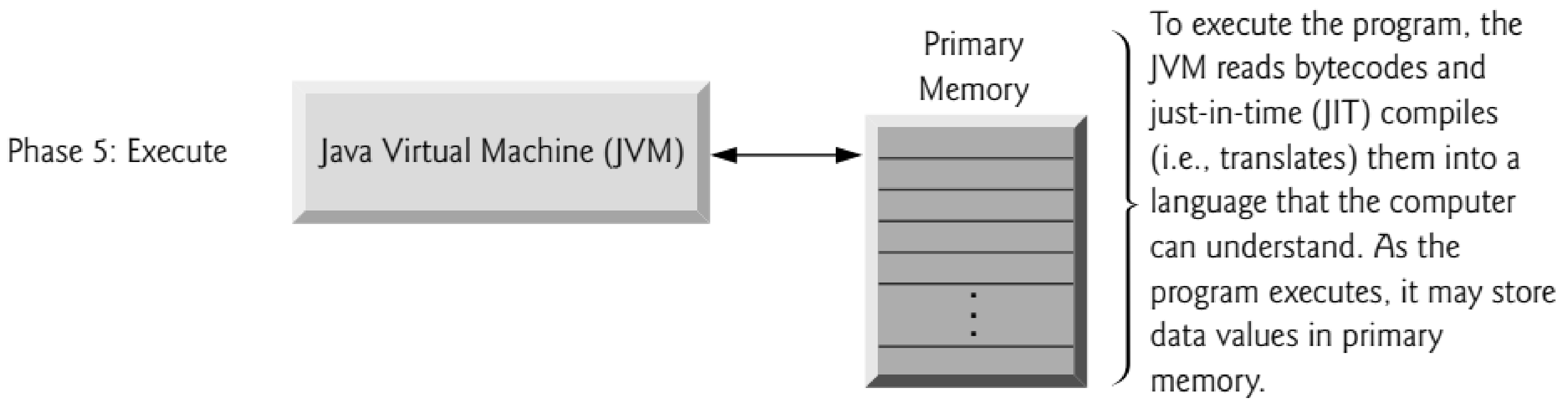

Phase 5: Execute
Java Virtual Machine (JVM)
Primary Memory
To execute the program, the JVM reads bytecodes and just-in-time (JIT) compiles (i.e., translates) them into a language that the computer can understand. As the program executes, it may store data values in primary memory.

Figure 1.9: Chapter 1 - (Java: How to program: Late Objects)

Java Development Environment

- *Phase 5: Execution*
  - *Two* compilation phases
    - One in which source code is translated into bytecodes.
    - Second in which, during execution, the *bytecodes* are translated into *machine language* for the actual computer on which the program executes

Java Development Environment

- **Common Programming Error**
  - *Errors such as division by zero occur as a program runs, so they're called runtime errors or execution-time errors.*
  - *Fatal runtime errors cause programs to terminate immediately without having successfully performed their jobs.*
  - *Nonfatal runtime errors allow programs to run to completion, often producing incorrect results.*

- Internet and World Wide Web

Internet and World Wide Web

Internet

Developed over three decades ago with DOD funding

Originally for connecting few main computer systems

Now accessible by hundreds of millions of computers

World Wide Web (WWW)

Allows for locating/viewing multimedia-based documents

- Introduction to Object Technology

Introduction to Object Technology

- Objects
  - Reusable software components that model real-world items
  - Look all around you
    - People, cars, mobile phone, etc.
  - Attributes
    - Size, shape, color, weight, etc.
  - Behaviors
    - Attend lecture, make call, stop engine, etc.

Introduction to Object Technology

- Object-oriented design (OOD)
  - Models real-world objects
  - Models communication among objects
  - *Encapsulates* data (attributes) and functions (behaviors)
    - Information hiding
    - Communication through well-defined interfaces
- Object-oriented language
  - Programming is called *object-oriented programming* (*OOP*)
  - Java

Tutorial to Install JDK and IDE

1. Install JDK:

- https://docs.oracle.com/en/java/javase/11/install/installation-jdk-microsoft-windows-platforms.html#GUID-A740535E-9F97-448C-A141-B95BF1688E6F

2. Install any of the following IDE:

- Netbeans
  - https://studyopedia.com/netbeans/installing-netbeans-ide-windows-10/
- BlueJ
  - https://www.bluej.org/download/install.html
- Eclipse
  - https://www.eclipse.org/downloads/packages/installer
- IntelliJ IDEA
  - https://www.jetbrains.com/help/idea/installation-guide.html#toolbox

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 3

Introduction to Java applications I

Contents

1. Writing First Program in Java (Block of Code, Statements, Keywords)
2. Displaying Text (print, println, printf, escape sequences)
3. Obtain User Input from the Keyboard
4. Import Libraries
5. Commenting Code
6. Debug and Trace Errors

Weekly Learning Outcomes

- Write first program in Java
- Manipulate the debugging tools and trace errors
- Show the use of Java libraries

Required Reading

1. Chapter 2 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 2 - (Java The Complete Reference, Eleventh Edition)
2. Importing Libraries:

http://www.mathcs.emory.edu/~cheung/Courses/170/Syllabus/04/java-lib.html

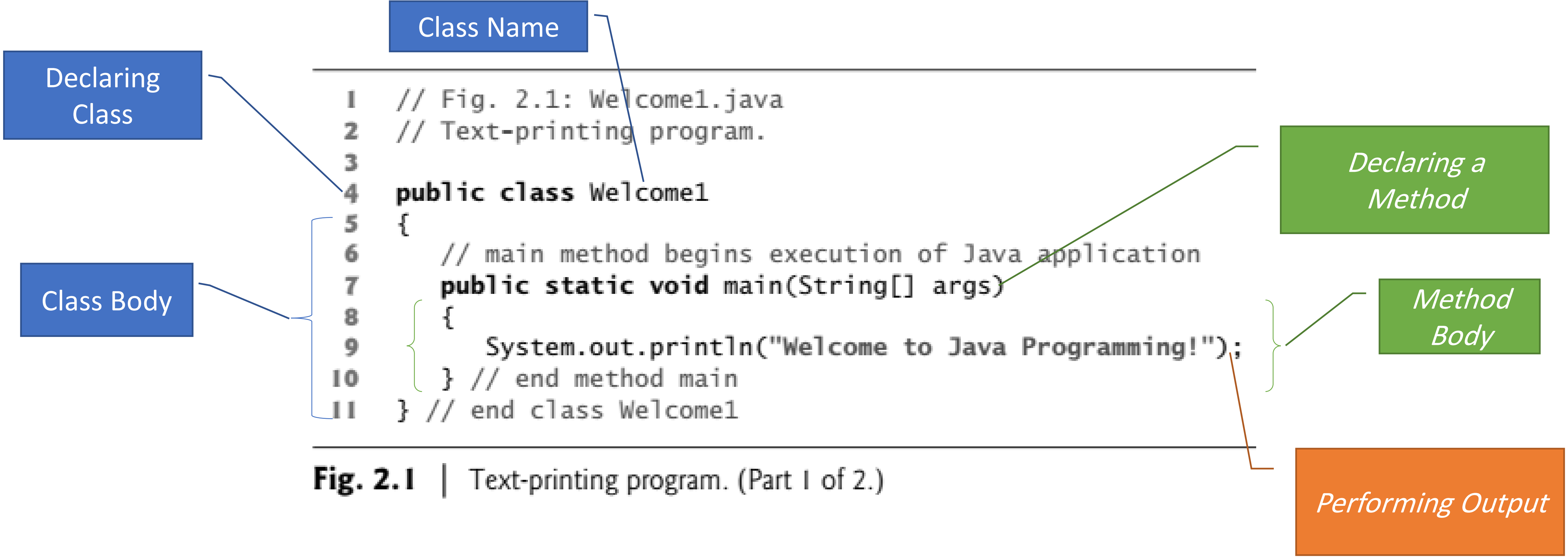
3. Debugging with Netbeans:

https://docs.oracle.com/javame/config/cdc/cdc-opt-impl/ojmeec/1.1/developer/html/debugging_netbeans.htm

- Writing First Program in Java (Block of Code, Statements, Keywords)

Writing First Program in Java

- A Java application is a computer program that executes when you use the **java** command to launch the Java Virtual Machine (JVM).



Class Name

Declaring Class

Class Body

Declaring a Method

Method Body

Performing Output

```java
// Fig. 2.1: Welcome1.java
// Text-printing program.

public class Welcome1
{
   // main method begins execution of Java application
   public static void main(String[] args)
   {
      System.out.println("Welcome to Java Programming!");
   } // end method main
} // end class Welcome1
```

Fig. 2.1 | Text-printing program. (Part 1 of 2.)

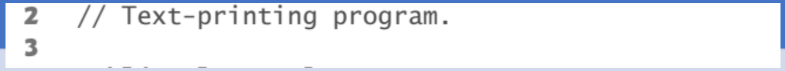
Writing First Program in Java

- *Commenting Your Programs:*
  - Comments ignored during program execution
  - To document programs
  - To improve their readability

1. One Line:
   - // Fig. 2.1: Welcome1.java
2. Multi-Lines:
   - /* This is a traditional comment. It
     can be split over multiple lines */
3. Javadoc comments
   - **/**** and ***/**
   - to embed program documentation directly in your programs

Writing First Program in Java

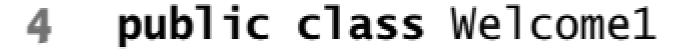
Blank line



```
// Text-printing program.

```

- Makes program more readable
- Blank lines, spaces, and tabs are white-space characters
    - Ignored by compiler

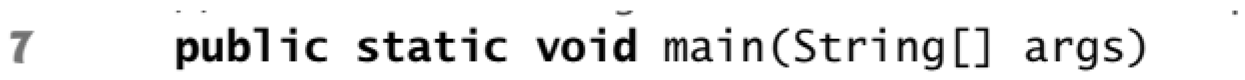
Begins class declaration for class Welcome1



```
public class Welcome1
```

- Every Java program has at least one user-defined class
- Keyword: words reserved for use by Java
    - class keyword followed by class name

Writing First Program in Java



```
public static void main(String[] args)
```

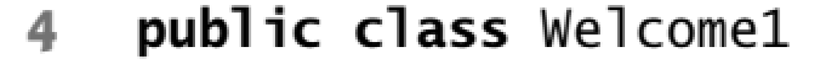
Name of class called identifier

- Series of characters consisting of letters, digits, underscores (_) and dollar signs ($)
- Does not begin with a digit, has no spaces
- Examples: Welcome1, **$**value, _value, button7
  - **7button** is invalid
- Java is case sensitive (capitalization matters)
  - a1 and A1 are different

public keyword will be discussed in later chapters in detail.

Writing First Program in Java

Saving files



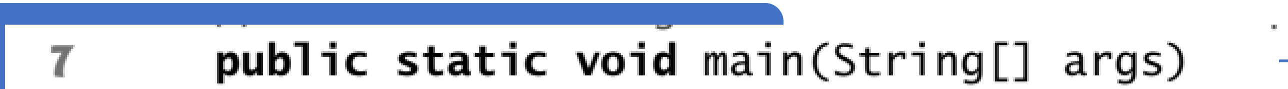
```
4    public class Welcome1
```

- File name must be class name with .java extension
- Welcome1.java

Left brace {

- Begins body of every class
- Right brace ends declarations (line 13)

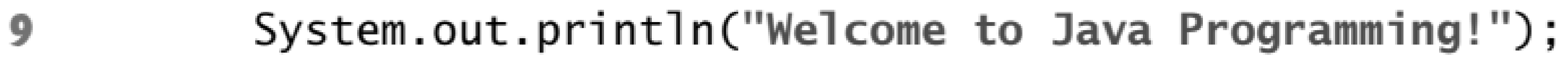
Part of every Java application



```
7         public static void main(String[] args)
```

- Applications begin executing at main
  - Parenthesis indicate main is a method (ch. 6)
  - Java applications contain one or more methods

Writing First Program in Java



```
        System.out.println("Welcome to Java Programming!");
```

Instructs computer to perform an action

- Prints string of characters
  - String - series characters inside double quotes
- White-spaces in strings are not ignored by compiler

System.out

- Standard output object
- Print to command window (i.e., MS-DOS prompt)

Method System.out.println

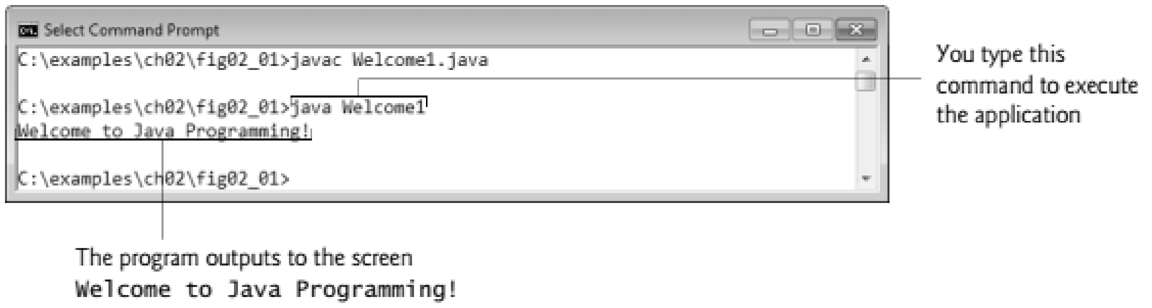
- Displays line of text
- Argument inside parenthesis

This line known as a statement

- Statements must end with semicolon ;

Writing First Program in Java

- *Compiling Your First Java Application*
  - ***javac Welcome1.java***
    - If no errors, `Welcome1.class` created
      - Has bytecodes that represent application
      - Bytecodes passed to Java interpreter
- *Executing the **Welcome1** Application*
  - ***java Welcome1***
    - Interpreter loads `.class` file for class `Welcome1`
    - `.class` extension omitted from command
    - Interpreter calls method `main`



```
C:\examples\ch02\fig02_01>javac Welcome1.java

C:\examples\ch02\fig02_01>java Welcome1
Welcome to Java Programming!

C:\examples\ch02\fig02_01>
```

You type this command to execute the application

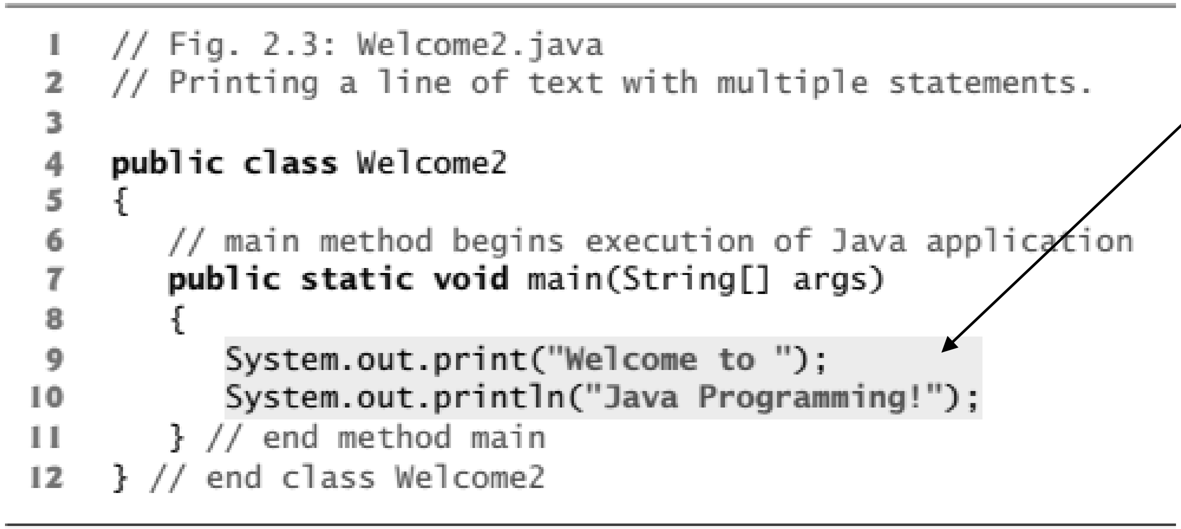
The program outputs to the screen `Welcome to Java Programming!`

Fig. 2.2 | Executing `Welcome1` from the **Command Prompt**.

Writing First Program in Java

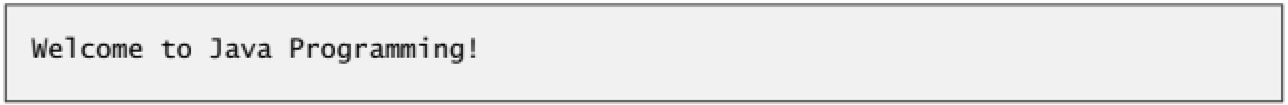
- **Modifying Your First Java Program**
  - *Displaying a Single Line of Text with Multiple Statements*

`System.out.print` keeps the cursor on the same line, so `System.out.println` continues the same line.



```java
// Fig. 2.3: Welcome2.java
// Printing a line of text with multiple statements.

public class Welcome2
{
   // main method begins execution of Java application
   public static void main(String[] args)
   {
      System.out.print("Welcome to ");
      System.out.println("Java Programming!");
   } // end method main
} // end class Welcome2
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 1 of 2.)

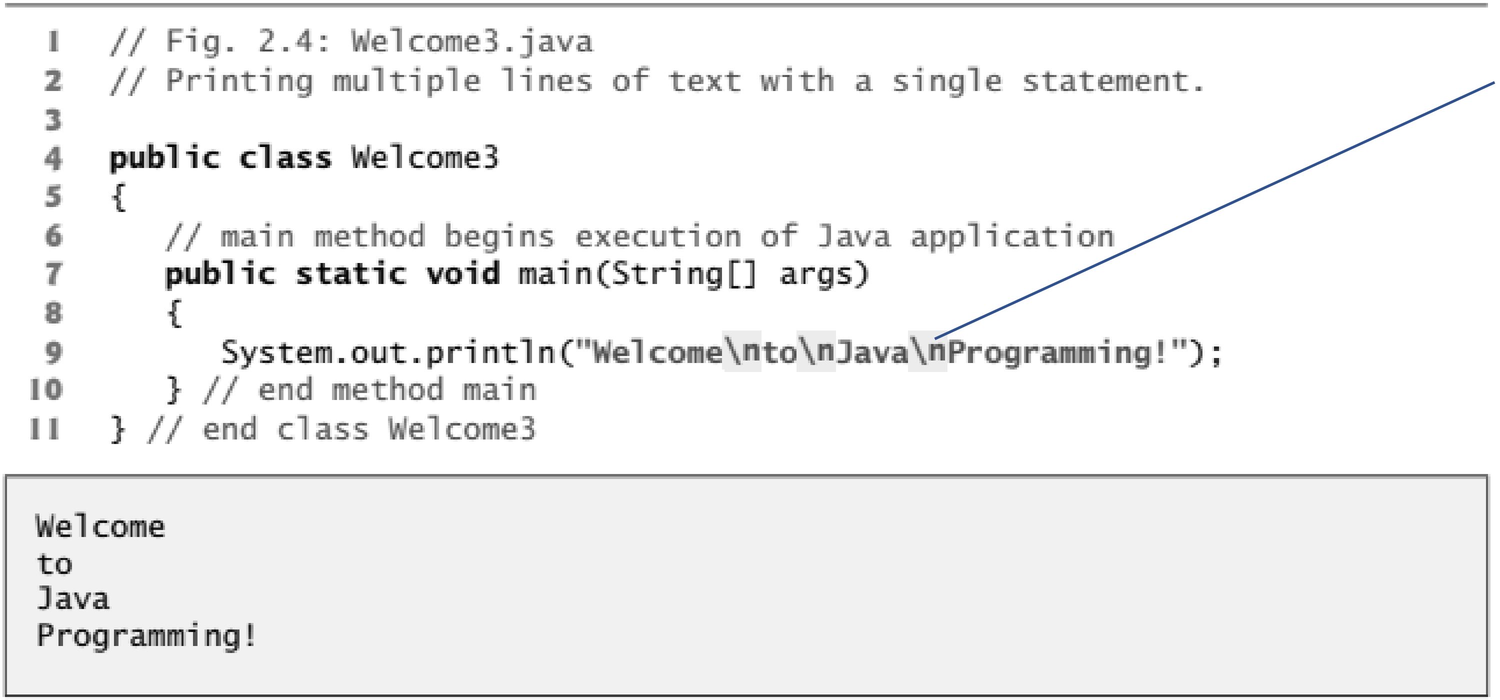


```
Welcome to Java Programming!
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 2 of 2.)

Writing First Program in Java

- *Displaying Multiple Lines of Text with a Single Statement*



```java
// Fig. 2.4: Welcome3.java
// Printing multiple lines of text with a single statement.

public class Welcome3
{
   // main method begins execution of Java application
   public static void main(String[] args)
   {
      System.out.println("Welcome\nto\nJava\nProgramming!");
   } // end method main
} // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

Newline character

Fig. 2.4 | Printing multiple lines of text with a single statement.

Writing First Program in Java

Displaying Multiple Lines of Text with a Single Statement

Newline characters (\n)

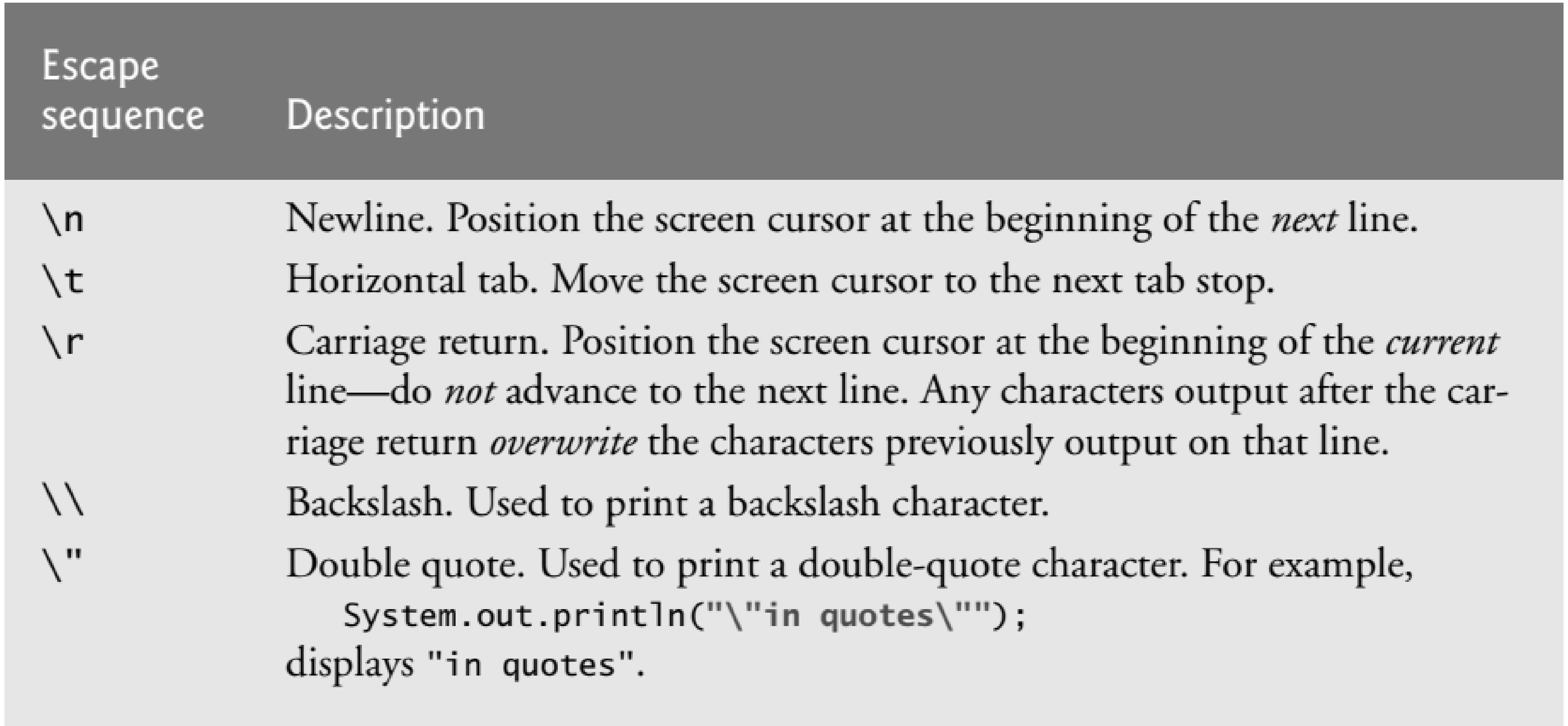
Interpreted as “special characters” by methods System.out.print and System.out.println

Indicates cursor should be on next line

Usage

Can use in System.out.println or System.out.print to create new lines

Writing First Program in Java



Escape sequence	Description
`\n`	Newline. Position the screen cursor at the beginning of the *next* line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor at the beginning of the *current* line—do *not* advance to the next line. Any characters output after the carriage return *overwrite* the characters previously output on that line.
`\\`	Backslash. Used to print a backslash character.
`\"`	Double quote. Used to print a double-quote character. For example, `System.out.println("\"in quotes\"");` displays `"in quotes"`.

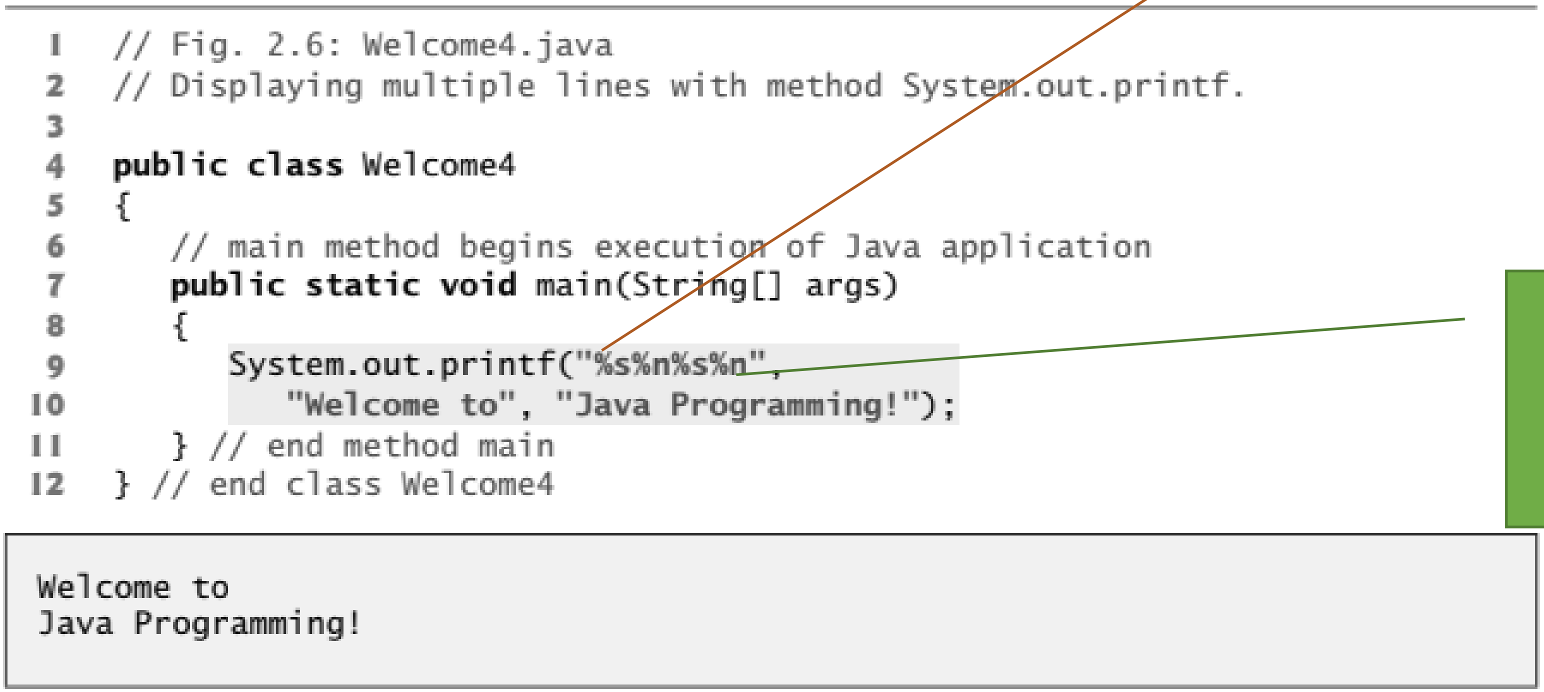
Fig. 2.5 | Some common escape sequences.

- Displaying Text

Displaying Text

- **Displaying Text with printf**
  - **printf** method (f means “formatted”) displays *formatted* data

the format specifier **%s** is a placeholder for a string



```java
// Fig. 2.6: Welcome4.java
// Displaying multiple lines with method System.out.printf.

public class Welcome4
{
   // main method begins execution of Java application
   public static void main(String[] args)
   {
      System.out.printf("%s%n%s%n",
         "Welcome to", "Java Programming!");
   } // end method main
} // end class Welcome4
```

```
Welcome to
Java Programming!
```

Fig. 2.6 | Displaying multiple lines with method `System.out.printf`.

the **%n** format specifier, which is a line separator

Displaying Text

- **Displaying Text with printf**
  - System.out.printf(format, arguments);

  - Example:
    - System.out.printf("Hello %s!%n", "World");
    - Output:
      - Hello World!
  - **Format Rules**
    - %[flags][width][.precision]conversion-character
    - **The *[flags]* define standard ways to modify the output**
    - ***The [width]* specifies the field width for outputting the argument.**
    - ***The [.precision]* specifies the number of digits of precision**

Displaying Text

- **Displaying Text with printf**
  - **Conversion Characters**
    - *s* formats strings.
    - *d* formats decimal integers.
    - *f* formats floating-point numbers.
    - *t* formats date/time values.

Displaying Text

- **Displaying Text with printf**
  - **Boolean Formatting**
    - System.out.printf("%b%n", **null**);
    - System.out.printf("%B%n", **false**);
    - System.out.printf("%B%n", 5.3);
    - System.out.printf("%b%n", "random text");

    - **Output:**
    - false
    - FALSE
    - TRUE
    - true

Displaying Text

- **Displaying Text with printf**
  - **String Formatting**
    - printf("'%s' %n", "baeldung");
    - printf("'%S' %n", "baeldung");
      - Output:
      - 'baeldung'
      - 'BAELDUNG’

    - printf("'%15s' %n", "baeldung");
      - Output:
      - '   baeldung'

Displaying Text

- **Displaying Text with printf**
  - **Number Formatting**
    - System.out.printf(Locale.US, "%,d %n", 10000);
    - System.out.printf(Locale.ITALY, "%,d %n", 10000);
      - Output:
      - 10,000
      - 10.000

    - System.out.printf("'%5.2f'%n", 5.1473);
      - Output:
      - ' 5.15'

Displaying Text

- **Displaying Text with printf**
  - **Date and Time Formatting**
    - System.out.printf("hours %tH: minutes %tM: seconds %tS%n", date, date, date);
      - Having used *H*, *M* and *S*, we get this result:
      - hours 13: minutes 51: seconds 15
    - System.out.printf("%1$tA, %1$tB %1$tY %n", date);
      - Then, using *A*, *B* and *Y*, we'd get this output:
      - Thursday, November 2018

- Obtain User Input from the Keyboard

Obtain User Input from the Keyboard

- When you read input from the user, you need to store it in your program.
  - A variable is a location in the computer’s memory where a value can be stored for use later in a program.
    - Variables will be discussed in details in the next lecture.

  - To read an input you use A **Scanner** to enables a program to read data
    - Scanner input = **new** Scanner(System.in);

    - This expression uses the **new** keyword to create a Scanner object that reads characters typed by the user at the keyboard.
    - The standard input object, **System.in**, enables applications to read *bytes* of data typed by the user.

Obtain User Input from the Keyboard

- **First you Prompt the User for Input:**
  - System.out.print(**"Enter first integer: "**); // prompt
- **Then Obtaining an int as Input from the User**
  - number1 = input.nextInt(); // read first number from user
    - nextInt method to obtain an integer from the user at the keyboard.

Obtain User Input from the Keyboard

- **Simple Example:**

```
import java.util.Scanner; // Import the Scanner class
class Main {
        public static void main(String[] args) {
                Scanner myObj = new Scanner(System.in); // Create a Scanner object
                System.out.println("Enter username");

                String userName = myObj.nextLine(); // Read user input
                System.out.println("Username is: " + userName); // Output user input
        }
}
```

Reference: https://www.w3schools.com/java/java_user_input.asp

- Import Libraries

Import Libraries

A great strength of Java is its rich set of predefined classes that you can *reuse* rather than "reinventing the wheel." These classes are grouped into packages.

referred to as the Java class library, or the Java Application Programming Interface (Java API)

```
import java.util.Scanner; // program uses class Scanner
```

- **import** declaration that helps the compiler locate a class that's used in this program
- the package that is named **java.util**

Import Libraries

- **Java API Documentation**
  - For each new Java API class we use, we indicate the package in which it’s located.
  - The Java API Documentation includes a detailed description of each package and its classes.
    - https://docs.oracle.com/javase/7/docs/api/index.html
  - You can download it from the Additional Resources section at
    - http://www.oracle.com/technetwork/java/javase/downloads/index.html

- Commenting Code

Commenting Code

- Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format.
  - Following is a simple example where the lines inside /*….*/ are Java multi-line comments. Similarly, the line which precedes // is Java single-line comment.

```
/**
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
*
* @author  name
* @version 1.0
* @since 2014-03-31
*/
public class HelloWorld {
        public static void main(String[] args) {
                // Prints Hello, World! on standard output.
                System.out.println("Hello World!");
        }
}
```

Reference: https://www.tutorialspoint.com/java/java_documentation.htm

Commenting Code

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator {
/** The add() method returns addition of given numbers.*/
public static int add(int a, int b){return a+b;}
/** The sub() method returns subtraction of given numbers.*/
public static int sub(int a, int b){return a-b;}
}
```

Compile it by javac tool:

```
javac Calculator.java
```

Create Documentation API by javadoc tool:

```
javadoc Calculator.java
```

- Now, there will be HTML files created for your Calculator class in the current directory

Reference: https://www.javatpoint.com/java-comments

- Debug and Trace Errors

Debug and Trace Errors

- **Debugging** is the routine process of locating and removing bugs, errors or abnormalities from programs.
    - **A conditional breakpoint:**
        - Applying breakpoints lets a debugger temporarily suspend the execution of your program at a certain point in the code.
        - This allows you to examine the state of the application, the values of local variables, the call stack, and to step through the code line-by-line.
        - This information gives you more clues to why the problem occurs.
        - To set a breakpoint, just toggle it by clicking on the left-hand side of the IDE screen.

Debug and Trace Errors

- **Exception breakpoints:**
  - Runtime exceptions like NullPointerException and ArrayIndexOutofBoundException are the errors encountered most often.
  - Figuring out the root cause of these errors can be difficult.
  - IDEs offer exception breakpoints in which the execution of the program is stopped whenever the specified exception occurs.
- **Watch point:**
  - It allows you to stop the execution of a program each time a targeted field or variable is accessed or changed.
  - The watchpoint can be applied on a field or variable.

Debug and Trace Errors

- **Check the following link for a complete tutorial for debugging steps in NetBeans:**
  - https://medium.com/@teguhteja/how-to-debugging-in-netbeans-java-c3087d9e2f6c

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

- https://www.baeldung.com/java-printstream-printf
- https://rollbar.com/guides/java/how-to-debug-java-errors/

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming
Week 4
Introduction to Java applications II

Contents

1. Variables
2. Memory Concepts
3. Operators
4. Increment/Decrement Operators
5. Assignment & Expressions
6. Logic and Syntax errors & Debugging concepts

Weekly Learning Outcomes

- Construct different variables types in Java
- Illustrate the usage of operators, assignments and expressions

Required Reading

1. Chapter 2 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 3, Chapter 4 - (Java The Complete Reference, Eleventh Edition)
2. Java Variables: https://www.javatpoint.com/java-data-types
3. Java Data Types: https://www.javatpoint.com/java-data-types

- Variables

Variables

Location in memory that stores a value

- Declare with ***name*** and ***type*** before use

A variable's ***name*** can be any valid identifier

Series of characters consisting of letters, digits, underscores (_) and dollar signs ($)

- Does not begin with a digit, has no spaces
- Examples: Welcome1, **$**value, _value, button7
  - **7button** is invalid
- Java is case sensitive (capitalization matters)
  - a1 and A1 are different

A variable's ***type*** specifies what kind of information is stored at that location in memory.

Variables

- Declarations end with semicolons ;
  - `int number1;`
- Can declare multiple variables of the same type at a time
- Use comma separated list
  - `int number1, number2;`
- Can add comments to describe purpose of variables
  - `int number1;    // first number to be stored`

Variables

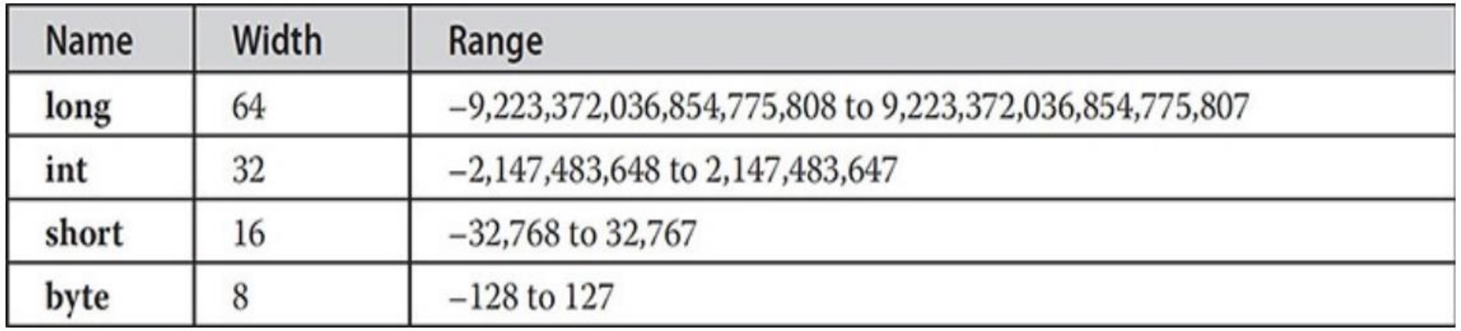
Variables Type (Primitive Types):

- Integers
  - can hold *integer* values (whole numbers such as 72, –1127 and 0)
  - The range of values for an int is –2,147,483,648 to +2,147,483,647
- Float & Double
  - hold real numbers
  - contain decimal points, such as in 3.4, 0.0 and –11.19
- Char
  - hold character data
  - represent individual characters, such as an uppercase letter (e.g., A), a digit (e.g., 7), a special character (e.g., * or %)
- Boolean
  - Hold True/False

Appendix D summarizes the characteristics of the eight primitive types (Boolean, byte, char, short, int, long, float and double).

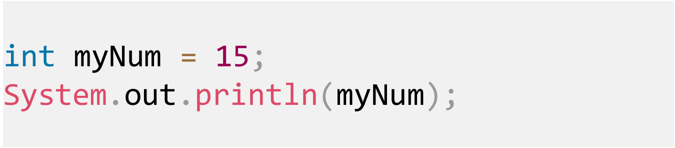
Variables

- Integers



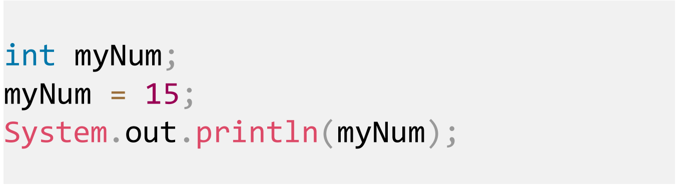
Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

- Declare a variable and assign value in one statement:



```
int myNum = 15;
System.out.println(myNum);
```

- Declare a variable and assign value in multiple statement:



```
int myNum;
myNum = 15;
System.out.println(myNum);
```

Variables

- Integers
  - Declare a variable and assign value, then change to a new value:

```
int myNum = 15;
myNum = 20; // myNum is now 20
System.out.println(myNum);
```

- Final Variables
  - this will declare the variable as "final" or "constant", which means unchangeable and read-only

```
final int myNum = 15;
myNum = 20; // will generate an error: cannot assign a value to a final variable
```

Variables

- Examples

```
int myNum = 5;               // Integer (whole number)
float myFloatNum = 5.99f;  // Floating point number
char myLetter = 'D’;       // Character
boolean myBool = true;     // Boolean
String myText = "Hello";   //String - Not primitive Data Type
```

Variables

- Display Variables:
    - String:

```java
String firstName = "John ";
String lastName = "Doe";
String fullName = firstName + lastName;
System.out.println(fullName);
```

- Integer:

```java
int x = 5; int y = 6;
System.out.println(x + y); // Print the value of x + y
```

- Memory Concepts

Memory Concepts

- Variable names such as number1 actually correspond to *locations* in the computer’s memory.
- Reading variables from memory does not change them
- The number typed by the user is placed into a memory location corresponding to the name number1.
  - int number1 = 45;

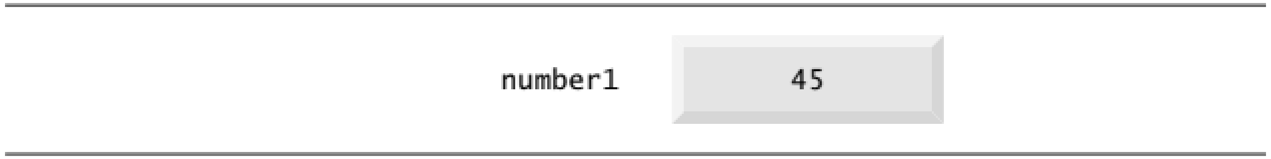
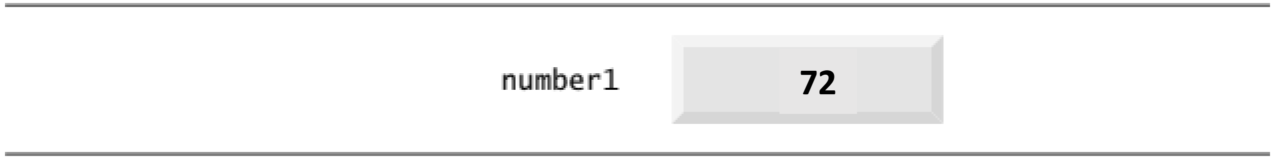


Fig. 2.8 | Memory location showing the name and value of variable `number1`.

- The computer places that integer value into location number1, replacing the previous value (if any) in that location.
  - number1 = 72;



- Operators

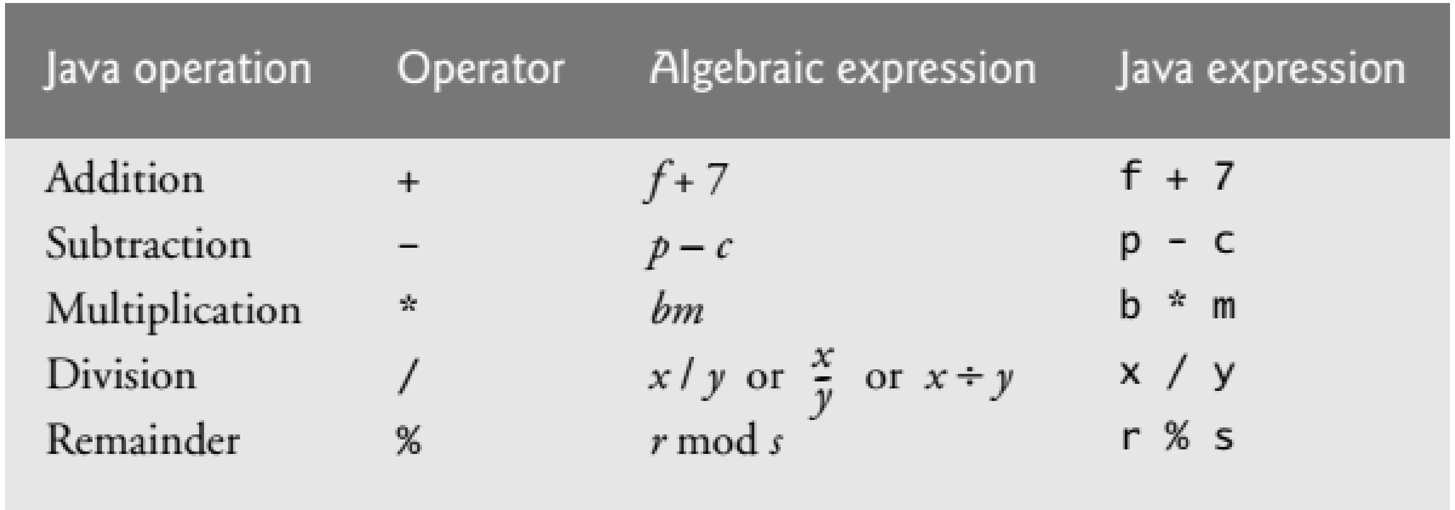
Operators

- Operators are used to perform operations on variables and values.
  - Example: + operator to add together two values
    - int number = 10 + 40;
  - Example:

```
int sum1 = 100 + 50; // 150 (100 + 50)
int sum2 = sum1 + 250; // 400 (150 + 250)
int sum3 = sum2 + sum2; // 800 (400 + 400)
```

Operators

- **Arithmetic**



Java operation	Operator	Algebraic expression	Java expression
Addition	+	$f + 7$	`f + 7`
Subtraction	–	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Division	/	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	%	$r \bmod s$	`r % s`

Fig. 2.11 | Arithmetic operators.

Operators

- **Arithmetic**
  - Integer division truncates remainder

    7 / 5 evaluates to 1
  - Remainder operator % returns the remainder

    7 % 5 evaluates to 2

Operators

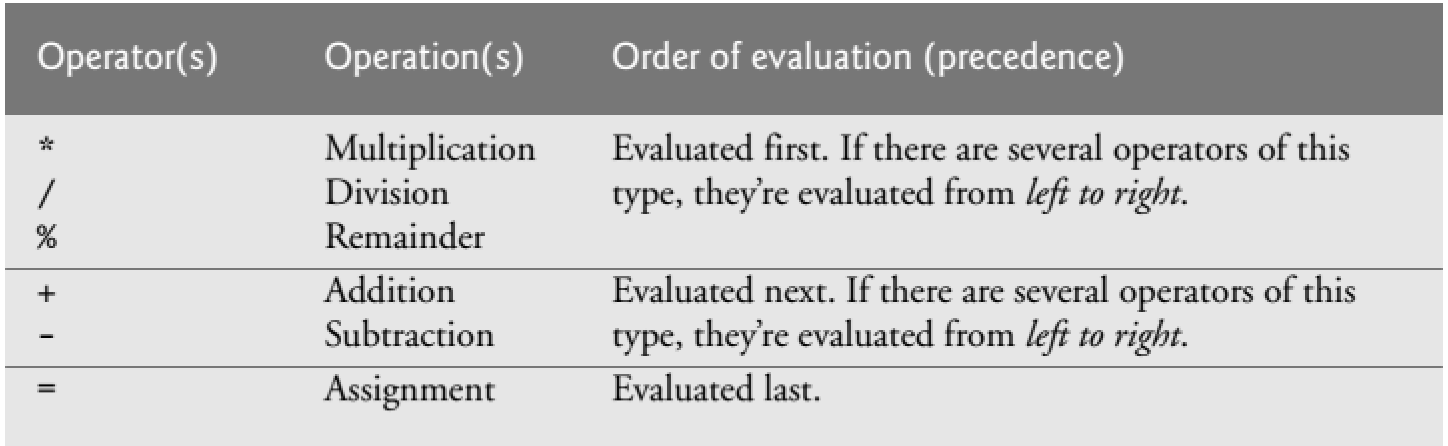
- **Arithmetic**
  - *Parentheses for Grouping Subexpressions*
    - a * (b + c)
      - It multiply a times the quantity b + c, we write

    - ((a + b) * c)
      - the expression in the *innermost* set of parentheses (a + b in this case) is evaluated *first*

    - Example:
      - ((3+2)*4)
      - (5 * 4)
      - 20

Operators

- **Arithmetic**
  - *Rules of Operator Precedence*
    - Some arithmetic operators act before others (i.e., multiplication before addition)
    - Multiplication, division and remainder operations are applied first.
      - If an expression contains several such operations, they’re applied from left to right.
    - Addition and subtraction operations are applied next.
      - If an expression contains several such operations, the operators are applied from left to right.



Operator(s)	Operation(s)	Order of evaluation (precedence)

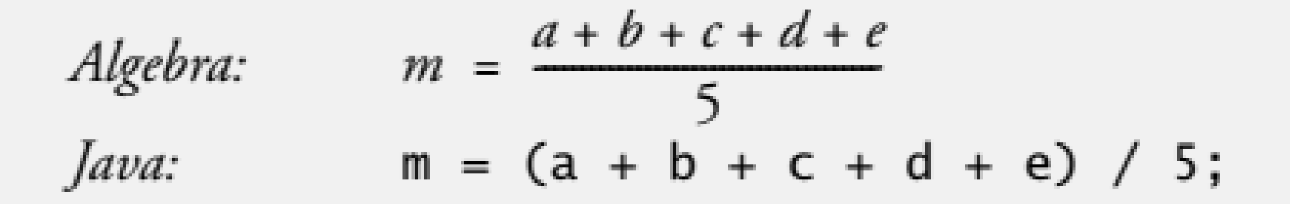
`*`
`/`
`%`
	Multiplication
	Division
	Remainder
		Evaluated first. If there are several operators of this type, they’re evaluated from *left to right*.

`+` `-`	Addition Subtraction	Evaluated next. If there are several operators of this type, they’re evaluated from *left to right*.
`=`	Assignment	Evaluated last.

Fig. 2.12 | Precedence of arithmetic operators.

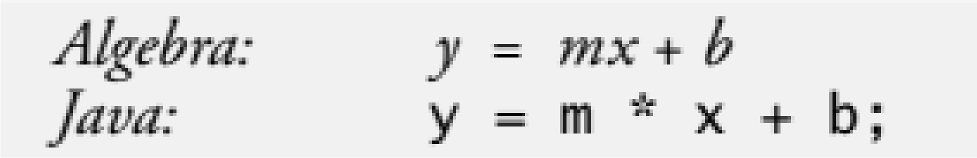
Operators

- **Arithmetic**
  - *Sample Algebraic and Java Expressions*



Algebra: $m = \frac{a + b + c + d + e}{5}$

Java: `m = (a + b + c + d + e) / 5;`

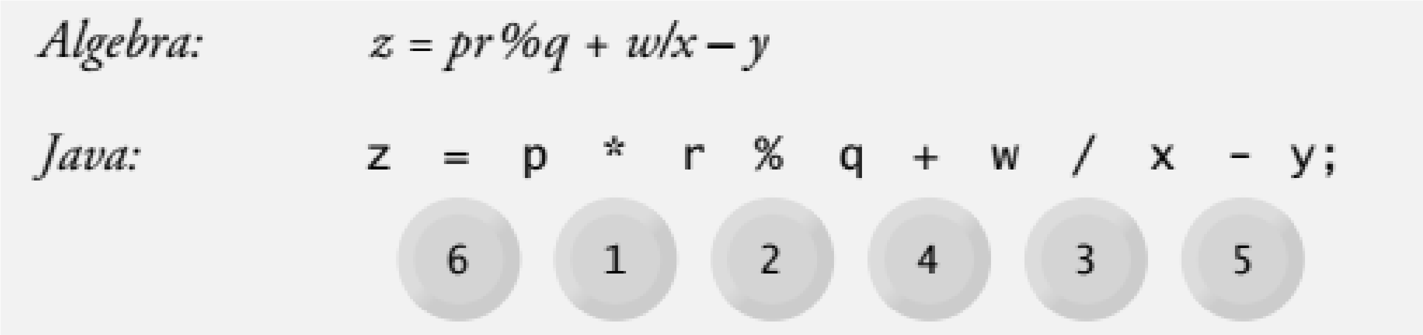


Algebra: $y = mx + b$

Java: `y = m * x + b;`

Operators

- **Arithmetic**
  - *Sample Algebraic and Java Expressions*



Algebra: $z = pr \% q + w/x - y$

Java: `z = p * r % q + w / x - y;`

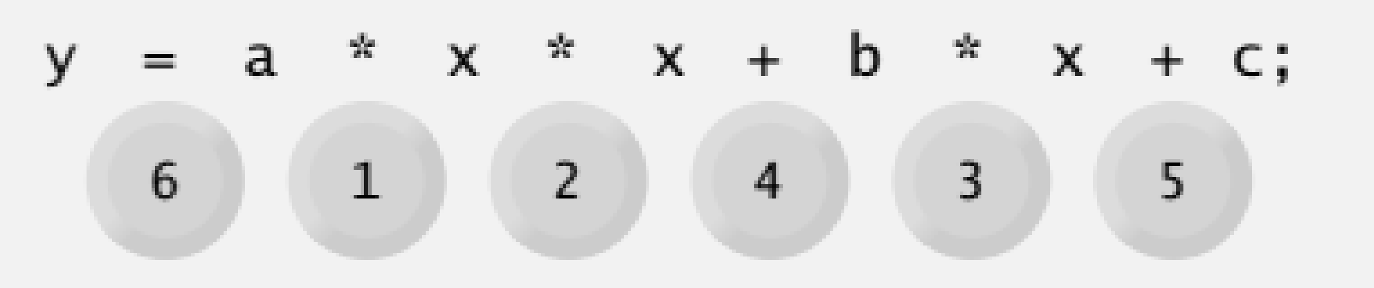
⑥ ① ② ④ ③ ⑤

- The circled numbers under the statement indicate the *order* in which Java applies the operators. The *, % and / operations are evaluated first in *left-to-right* order.

Operators

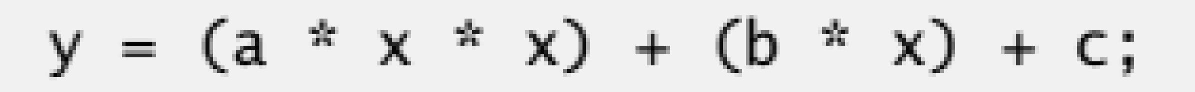
- **Arithmetic**
  - *Sample Algebraic and Java Expressions*
    - *Evaluation of a Second-Degree Polynomial*

$$ax^2 + bx + c$$



```
y = a * x * x + b * x + c;
```

- More simpler writing:



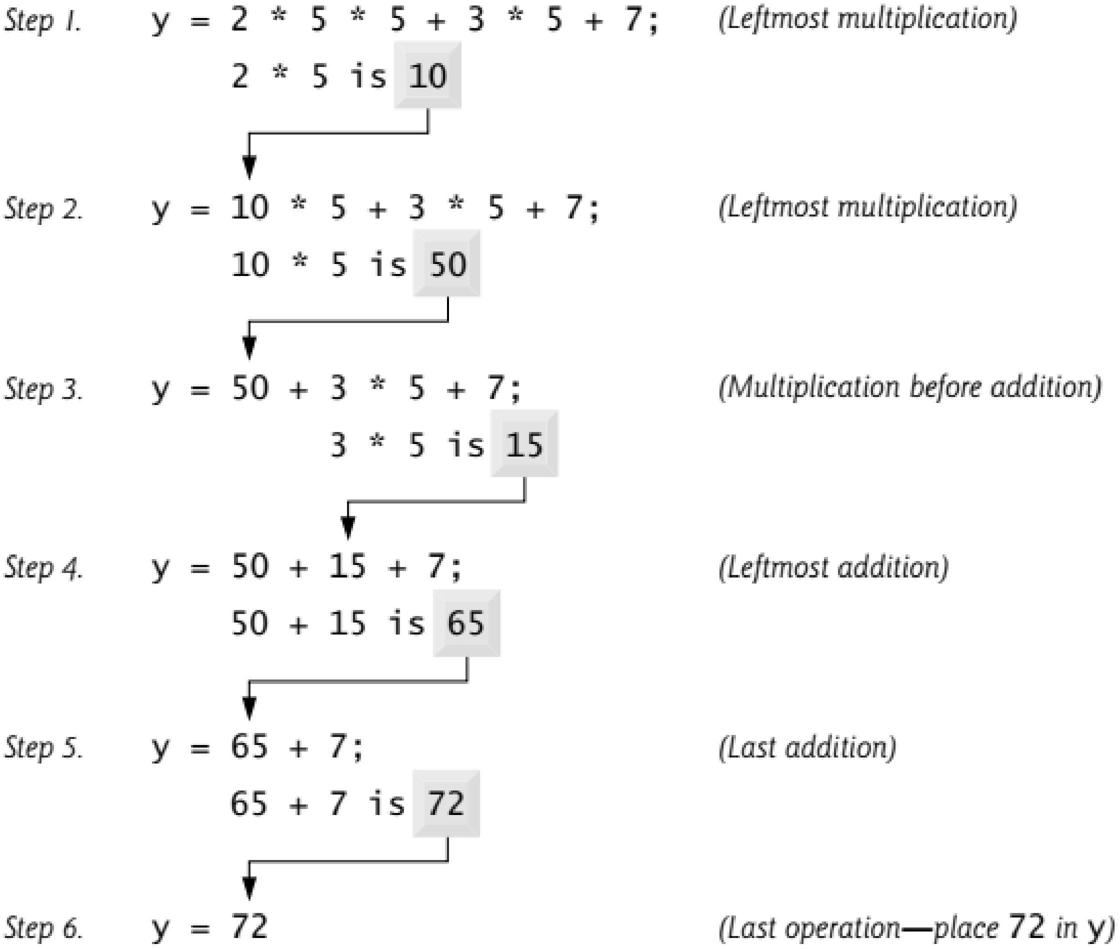
```
y = (a * x * x) + (b * x) + c;
```

Operators

- **Arithmetic**
  - *Evaluate this equation:*

```
y = (a * x * x) + (b * x) + c;
```

  - *With the following values* a = 2, b = 3, c = 7 and x = 5.



Step 1. `y = 2 * 5 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`2 * 5 is 10`

Step 2. `y = 10 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`10 * 5 is 50`

Step 3. `y = 50 + 3 * 5 + 7;` *(Multiplication before addition)*

`3 * 5 is 15`

Step 4. `y = 50 + 15 + 7;` *(Leftmost addition)*

`50 + 15 is 65`

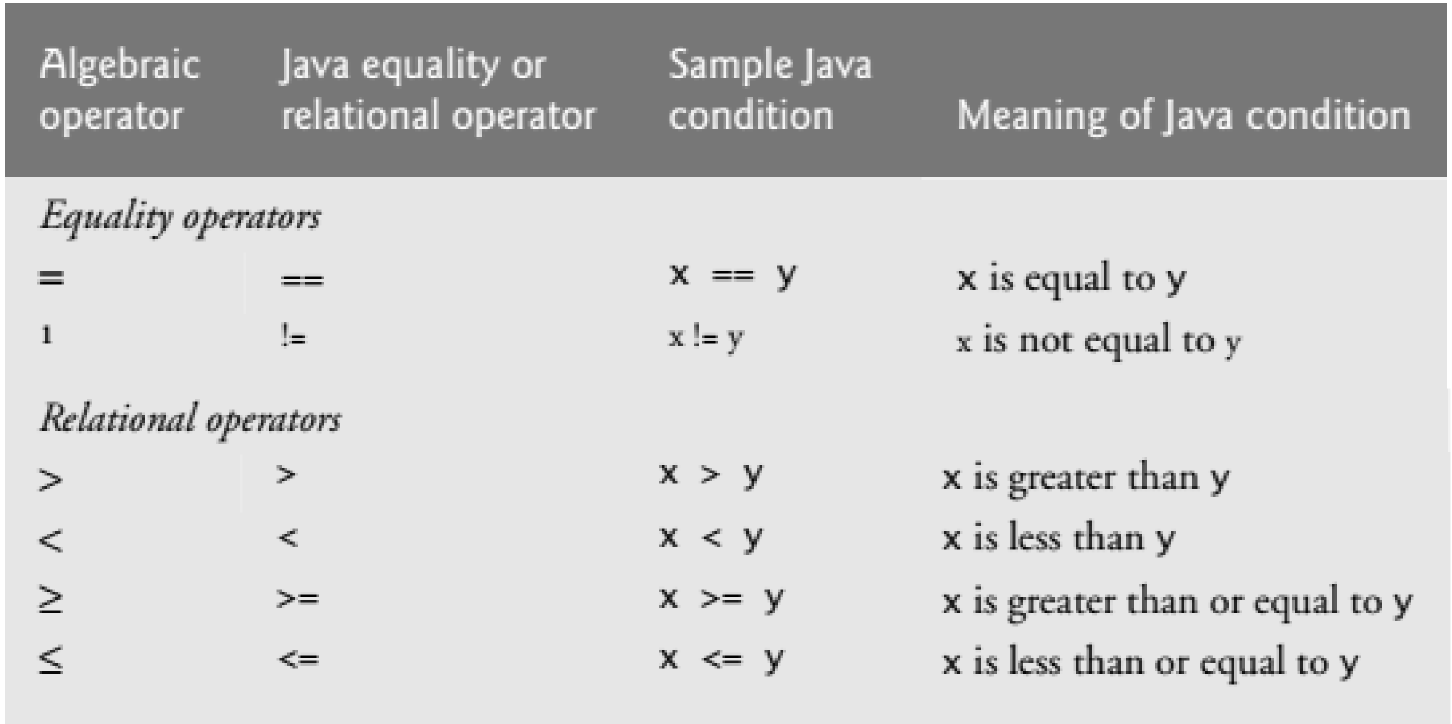
Step 5. `y = 65 + 7;` *(Last addition)*

`65 + 7 is 72`

Step 6. `y = 72` *(Last operation—place 72 in y)*

Operators

- **Equality and Relational Operators**
  - A condition is an expression that can be **true** or **false**.



Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
1	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Operators

- **Equality and Relational Operators**
  - Example:

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 3;
    System.out.println(x == y); // returns false because 5 is
not equal to 3
  }
}
```

  - Output: **False**

Operators

- **Equality and Relational Operators**
  - Example:

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 3;
    System.out.println(x > y); // returns true because 5 is
greater than 3
  }
}
```

  - Output: **True**

Operators

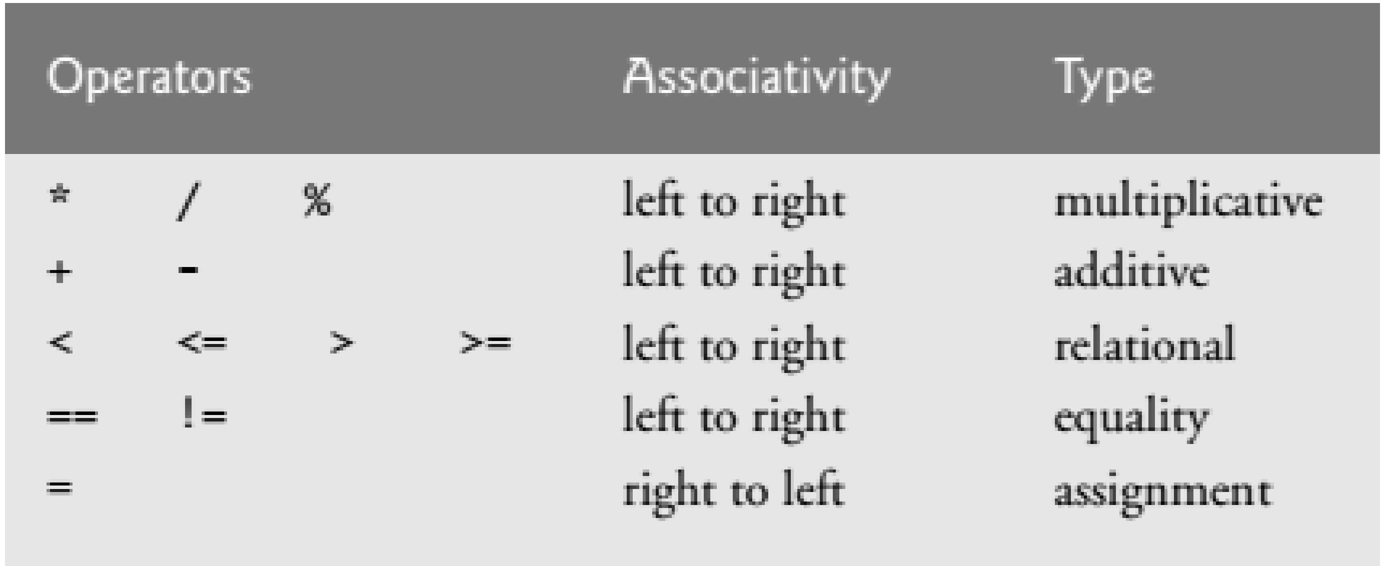
- **Equality and Relational Operators**
  - Example:

```
public class Main {
  public static void main(String[] args) {
    int x = 5;
    int y = 3;
    System.out.println(x <= y); // returns false because 5 is
neither less than or equal to 3
  }
}
```

  - Output: **False**

Operators

- **Operators Precedence**



Operators	Associativity	Type
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`=`	right to left	assignment

Fig. 2.16 | Precedence and associativity of operators discussed.

- Increment/Decrement Operators

Increment/Decrement Operators

++	**Increment**	**Increases the value of a variable by 1**

- The **pre increment** operator is used to increment the value of some variable before using it in an expression.

```
public class IncDec {
      public static void main(String[] args) {
            int a, b = 15;
            a = ++b;
            System.out.println(“” + a);
      }
}
```

- Output: **16**

Increment/Decrement Operators

++	**Increment**	**Increases the value of a variable by 1**

- In the **post increment** the value is used inside the expression, then incremented by one.

```
public class IncDec {
      public static void main(String[] args) {
            int a, b = 15;
            a = b++;
            System.out.println(“” + a);
      }
}
```

- Output: **15**

Increment/Decrement Operators

--	Decrement	Decreases the value of a variable by 1

- If a decrement operator is used in front of an operand, then it is called Pre decrement operator.

```
class PreDecrement {
        public static void main(String[] args) {
                int x = 10;
                int y = --x;
                System.out.println("y value is: " + y);
        }
}
```

- Output: **y value is: 9**

Increment/Decrement Operators

--	Decrement	Decreases the value of a variable by 1

- If a decrement operator is used after an operand, then it is called Post decrement operator.

```
class PostDecrement {
      public static void main(String[] args) {
            int x = 10;
            int y = x--;
            System.out.println("y value is: " + y);
      }
}
```

- Output: **y value is: 10**

- Assignment & Expressions

Assignment & Expressions

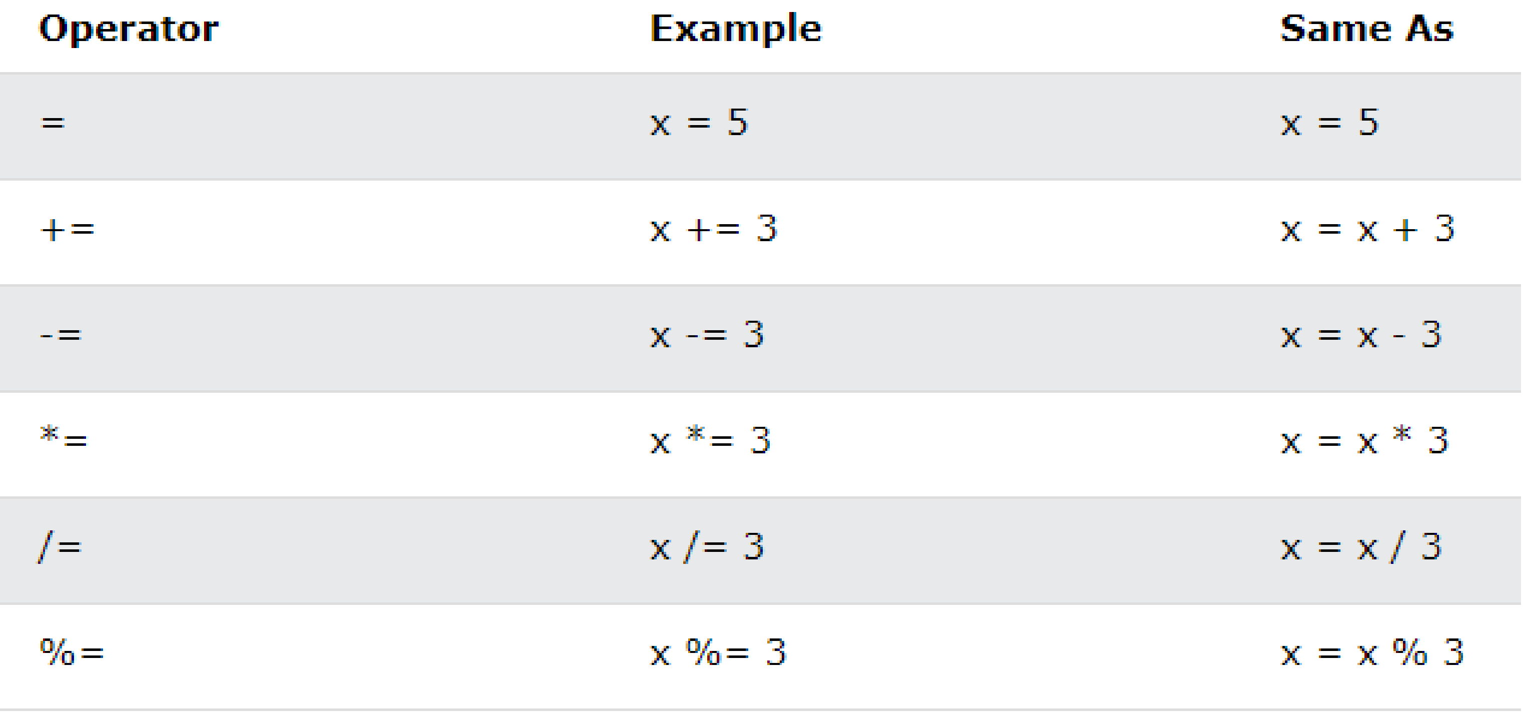
we use the assignment operator (=) to assign the value 10 to a variable called x:

- int x = 10;

The addition assignment operator (+=) adds a value to a variable:

- int x = 10;
- x += 5;

Assignment & Expressions



Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Reference: https://www.w3schools.com/java/java_operators.asp

- Logic and Syntax errors & Debugging concepts

Logic and Syntax errors & Debugging concepts

- Syntax errors

- These are errors where the compiler finds something wrong with your program, and you can't even try to execute it.
- For example, you may have incorrect punctuation, or may be trying to use a variable that hasn't been declared.

Logic and Syntax errors & Debugging concepts

- Syntax errors
  - Missing Parentheses
    - System.out.print;
  - Missing Semicolumn
    - int x=10

Logic and Syntax errors & Debugging concepts

Runtime errors

If there are no syntax errors, Java may detect an error while your program is running.

You will get an error message telling you the kind of error

Logic and Syntax errors & Debugging concepts

- Runtime errors (Divide by Zero Error)

```
class DivByZero {
    public static void main(String args[])
    {
        int var1 = 15;
        int var2 = 5;
        int var3 = 0;
        int ans1 = var1 / var2;

        // This statement causes a runtime error,
        // as 15 is getting divided by 0 here
        int ans2 = var1 / var3;

        System.out.println(
            "Division of va1"
            + " by var2 is: "
            + ans1);
        System.out.println(
            "Division of va1"
            + " by var3 is: "
            + ans2);
    }
}
```

Logic and Syntax errors & Debugging concepts

Logic errors

A logic error, or bug, is when your program compiles and runs, but does the wrong thing.

The Java system, of course, has no idea what your program is supposed to do, so it provides no additional information to help you find the error.

Logic and Syntax errors & Debugging concepts

- Logic errors
  - **Using incorrect operator precedence**

```
public class OperatorError {
        public static void main(String[] args) {
                // Create some variables.
                int MyVar1 = 5 + 4 * 3 / 2;
                int MyVar2 = (5 + 4) * 3 / 2;
                int MyVar3 = (5 + 4) * (3 / 2);
                int MyVar4 = (5 + (4 * 3)) / 2;
                // Output the result.
                System.out.println( "MyVar1: " + MyVar1 + "nMyVar2: " + MyVar2 + "nMyVar3: " +
MyVar3 + "nMyVar4: " + MyVar4);
        }

}
```

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230

Object Oriented Programming

Week 5

Control Statements I

Contents

1. Algorithms and Pseudocode
2. If statement
3. If….Else
4. Nested If Statement
5. Switch…..Case
6. Conditional Operator

Weekly Learning Outcomes

- Recognize the control statements for decision making
- Demonstrate the usage of nested if and switch statements

Required Reading

1. Chapter 3 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 5 - (Java The Complete Reference, Eleventh Edition)
2. Java if...else Statement: https://www.programiz.com/java-programming/if-else-statement
3. The switch Statement: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

- Algorithms and Pseudocode

Algorithms and Pseudocode

- A *procedure* for solving a problem in terms of
  1. the actions to execute and
  2. the order in which these actions execute
  - called an **algorithm**.

Algorithms and Pseudocode

- **Algorithm Example: Add two numbers entered by the user**

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
sum←num1+num2
Step 5: Display sum Step 6: Stop
```

Reference: https://www.programiz.com/dsa/algorithm

Algorithms and Pseudocode

- **Pseudocode** is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax.
- **Pseudocode** does not execute on computers.
-

Algorithms and Pseudocode

- **Pseudocode** Example:

```
If student's grade is greater than or equal to 60
    Print "passed"
else
    Print "failed"
```

Control Structure

- Sequential execution
  - Program statements execute one after the other

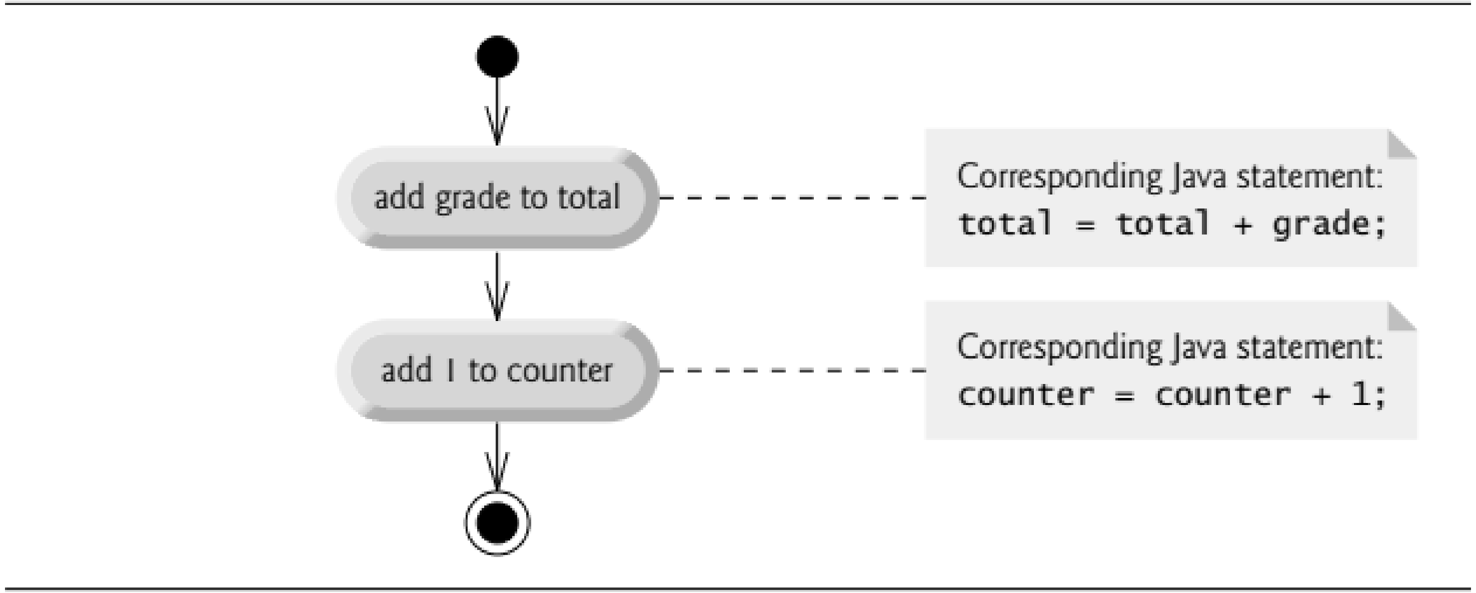


Fig. 3.1 | Sequence-structure activity diagram.

Control Structure

- Transfer of control
  - Three control statements can specify order of statements
    - Sequence structure
    - Selection structure
    - Repetition structure

Control Structure

Transfer of control

- Java has a sequence structure “built-in”
- Java provides three selection structures
  - if
  - If…else
  - switch
- Java provides three repetition structures
  - while
  - do…while
  - do

Each of these words is a Java keyword

- If statement

If statement

- Single-entry/single-exit control structure
- Perform action only when condition is `true`

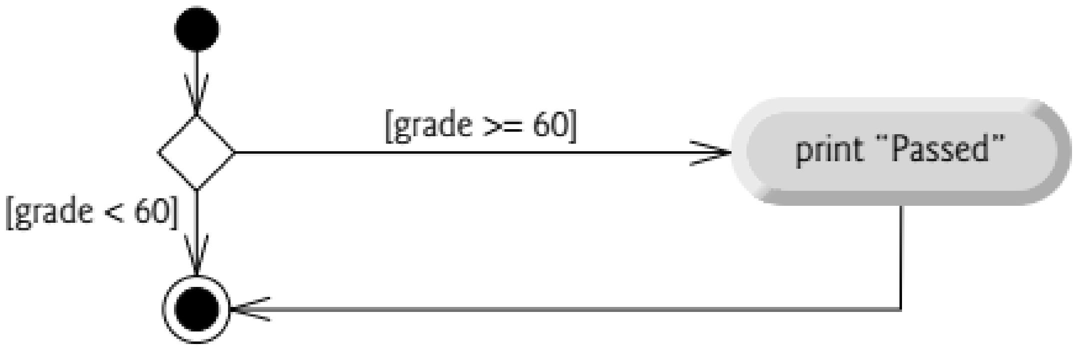


Fig. 3.2 | `if` single-selection statement UML activity diagram.

If statement

- Pseudocode

  ```
  If student’s grade is greater than or equal to 60
      Print “Passed
  ```

- Java Code

  ```java
  if (studentGrade >= 60)
      System.out.println("Passed");
  ```

If statement

- Example:

```
//Java Program to demonstate the use of if statement.
public class IfExample {
public static void main(String[] args) {
    //defining an 'age' variable
    int age=20;
    //checking the age
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Reference: https://www.javatpoint.com/java-if-else

- If….Else

If….Else

- Perform action only when condition is `true`
- Perform different specified action when condition is `false`

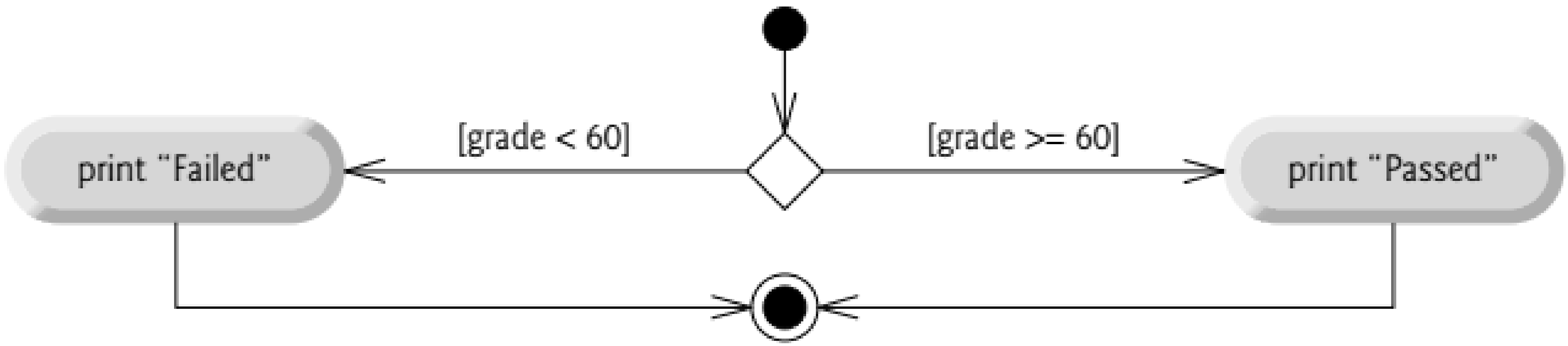


Fig. 3.3 | `if...else` double-selection statement UML activity diagram.

If….Else

- Pseudocode

```
If student’s grade is greater than or equal to 60
    Print “Passed”
Else
    Print “Failed
```

- Java Code

```
if (grade >= 60)
    System.out.println("Passed");
else
    System.out.println("Failed");
```

If….Else

- Example:

```
public class IfElseExample {
public static void main(String[] args) {
   //defining a variable
   int number=13;
   //Check if the number is divisible by 2 or not
   if(number%2==0){
      System.out.println("even number");
   }else{
      System.out.println("odd number");
   }
}
}
```

Reference: https://www.javatpoint.com/java-if-else

- Nested If Statement

Nested If Statement

- A program can test multiple cases by placing if…else statements inside other if…else statements to create nested **if**…**else** statements.

- Pseudocode

```
If student’s grade is greater than or equal to 90
        Print “A”
else
        If student’s grade is greater than or equal to 80
                Print “B”
        else
                If student’s grade is greater than or equal to 70
                        Print “C”
                else
                        If student’s grade is greater than or equal to 60
                                Print “D”
                        else
                                Print “F”
```

Nested If Statement

- Java Code

```
if (studentGrade >= 90)
    System.out.println("A");
Else
    if (studentGrade >= 80)
        System.out.println("B");
    Else
        if (studentGrade >= 70)
            System.out.println("C");
        Else
            if (studentGrade >= 60)
                System.out.println("D");
            Else
                System.out.println("F");
```

Nested If Statement

- Simplify the Java Code

```
if (studentGrade >= 90)
        System.out.println("A");
else if (studentGrade >= 80)
        System.out.println("B");
else if (studentGrade >= 70)
        System.out.println("C");
else if (studentGrade >= 60)
        System.out.println("D");
else
        System.out.println("F");
```

Nested If Statement

- *Dangling-**else** Problem*
- Java compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces ({ and }).

```
if (x > 5)
        if (y > 5)
                System.out.println("x and y are > 5");
else
        System.out.println("x is <= 5");
```

- The compiler will interpret it as the following:

```
if (x > 5)
        if (y > 5)
                System.out.println("x and y are > 5");
        else
                System.out.println("x is <= 5");
```

Nested If Statement

- *Dangling-**else** Problem*
- To force the nested if…else statement to execute as it was originally intended.

```
if (x > 5)
{
        if (y > 5)
                System.out.println("x and y are > 5");
}
else
        System.out.println("x is <= 5");
```

Nested If Statement

- *Blocks*
- The if statement normally expects only *one* statement in its body.
- To include *several* statements in the body of an if enclose the statements in braces.

```
if (grade >= 60)
        System.out.println("Passed");
else
{
        System.out.println("Failed");
        System.out.println("You must take this course again.");
}
```

- Switch.....Case

Switch…..Case

The **switch** multiple-selection statement performs different actions based on the possible values of a constant integral expression

The case values must be *unique*.

The Java switch expression must be of *byte, short, int, long, enums and string*.

Each case statement can have a *break statement* which is optional.

- It jumps the control after the switch expression.
- If a break statement is not found, it executes the next case.

The case value can have a *default label* which is optional.

Switch.....Case

```
switch(expression){
        case value1:
                //code to be executed;
                break;  //optional
        case value2:
                //code to be executed;
                break;  //optional
        ......

        default:
                code to be executed if all cases are not matched;
}
```

Reference: https://www.javatpoint.com/java-switch

Switch.....Case

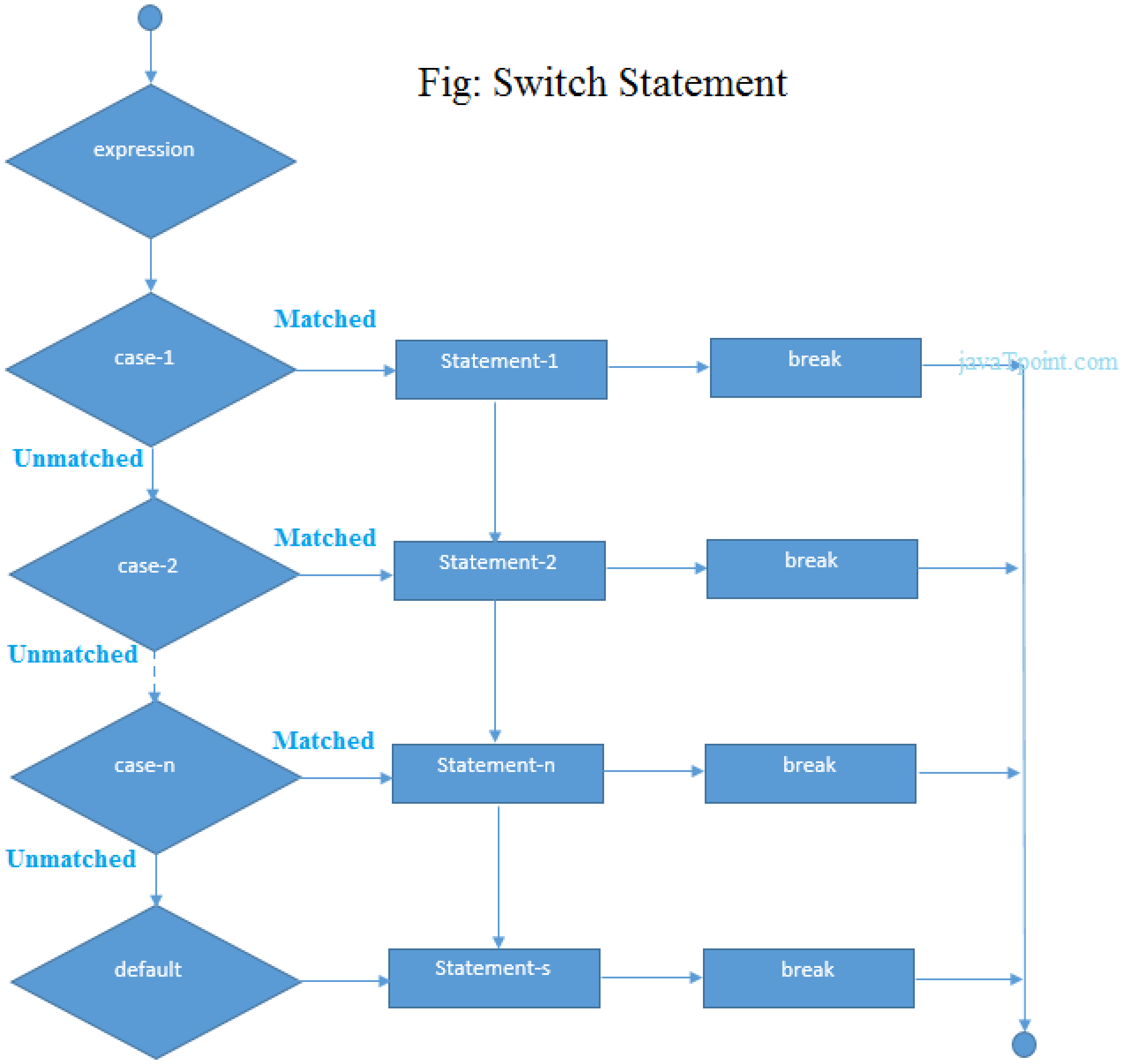


Fig: Switch Statement

Reference: https://www.javatpoint.com/java-switch

Switch…..Case

- Example:

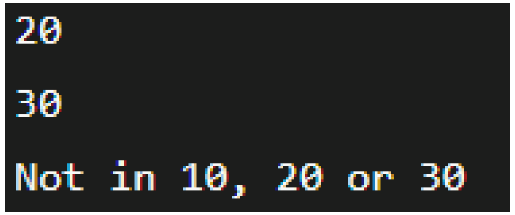
```
public class SwitchExample {
        public static void main(String[] args) {
                //Declaring a variable for switch expression
                int number=20;
                //Switch expression
                switch(number){
                //Case statements
                        case 10: System.out.println("10");
                                break;
                        case 20: System.out.println("20");
                                break;
                        case 30: System.out.println("30");
                                break;
                //Default case statement
                        default:System.out.println("Not in 10, 20 or 30");
                }
        }
}
```

Reference: https://www.javatpoint.com/java-switch

Switch…..Case

- The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

```
public class SwitchExample2 {
        public static void main(String[] args) {
                int number=20;
                //switch expression with int value
                switch(number){
                //switch cases without break statements
                        case 10: System.out.println("10");
                        case 20: System.out.println("20");
                        case 30: System.out.println("30");
                        default:System.out.println("Not in 10, 20 or 30");
                }
        }
}
```



```
20
30
Not in 10, 20 or 30
```

Reference: https://www.javatpoint.com/java-switch

Switch…..Case

- Example:

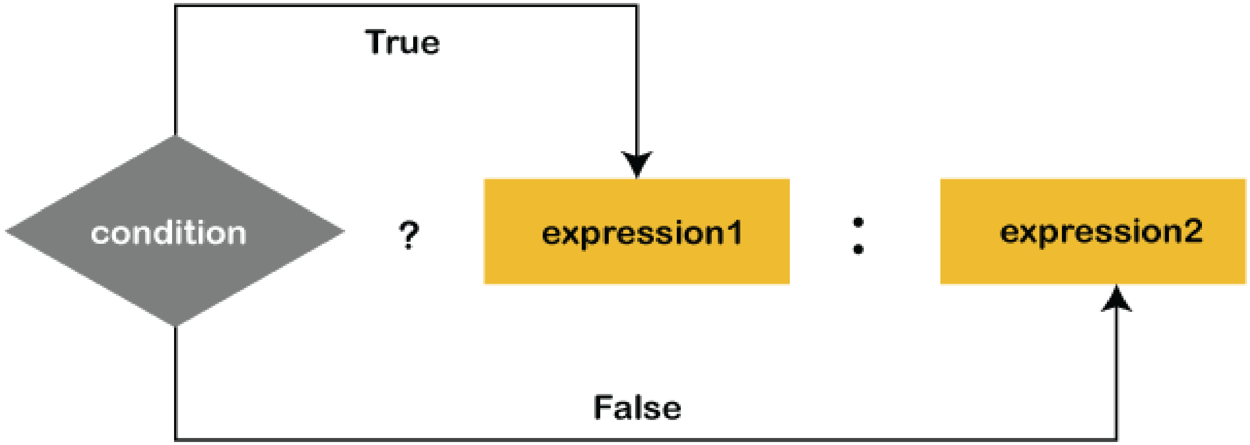
```
public class Test {
      public static void main(String args[]) {
      char grade = 'C’;
      switch(grade) {
            case 'A' : System.out.println("Excellent!");
                  break;
            case 'B’ :
            case 'C' : System.out.println("Well done");
                  break;
            case 'D' : System.out.println("You passed");
            case 'F' : System.out.println("Better try again");
                  break;
            default : System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
} }
```

Reference: https://www.tutorialspoint.com/java/switch_statement_in_java.htm

- Conditional Operator

Conditional Operator (?:)

- It can be used in place of an if…else statement.
- This can make your code shorter and clearer.
- The conditional operator is Java’s only ternary operator.



Reference: https://www.javatpoint.com/conditional-operator-in-java

Conditional Operator (?:)

- Example:

```
public class condiop {
        public static void main(String[] args) {
                String out;
                int a = 6, b = 12;
                out = a==b ? "Yes":"No";
                System.out.println("Ans: "+out);
        }
}
```

Reference: https://www.w3schools.in/java-tutorial/operators/conditional/

Conditional Operator (?:)

- Example:

```
public class LargestNumberExample  {
        public static void main(String args[])  {
                int x=69;
                int y=89;
                int z=79;
                int largestNumber= (x > y) ? (x > z ? x : z) : (y > z ? y : z);
                System.out.println("The largest numbers is:  "+largestNumber);
        }
}
```

Reference: https://www.javatpoint.com/conditional-operator-in-java

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 6

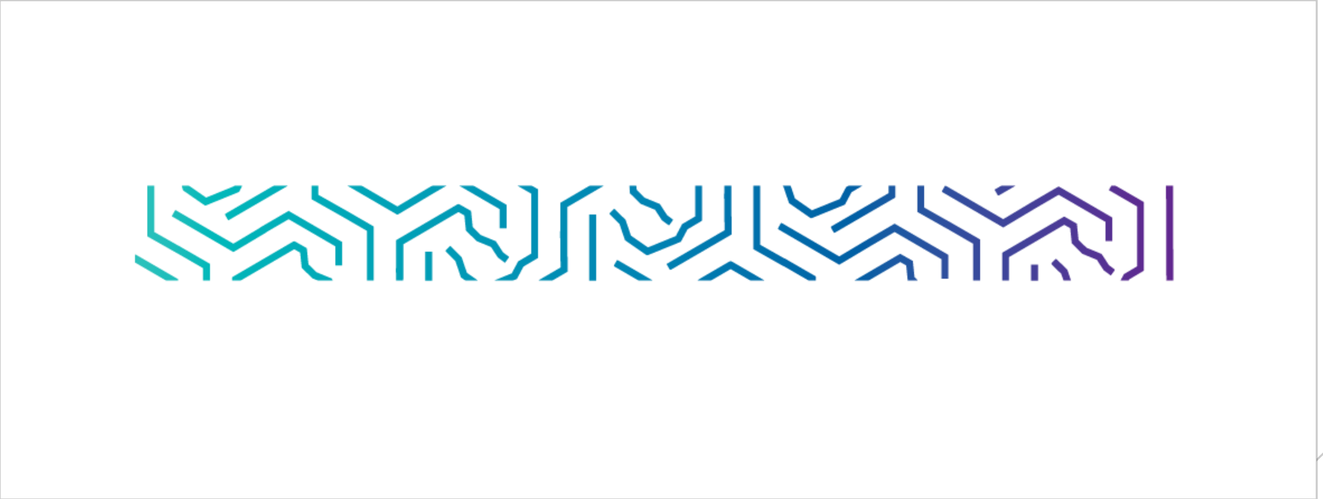
Control Statements II

Contents

1. For / While Loops
2. Nested Loops
3. Break and Continue Statements
4. Logical Operator

Weekly Learning Outcomes

- Operate different Loops statements techniques
- Differentiate between Break and Continue statements



Required Reading

1. Chapter 4 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 5 - (Java The Complete Reference, Eleventh Edition)
2. Java Loops Tutorial: https://www.learnjavaonline.org/en/Loops
3. Branching Statements:

   https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

- For / While Loops

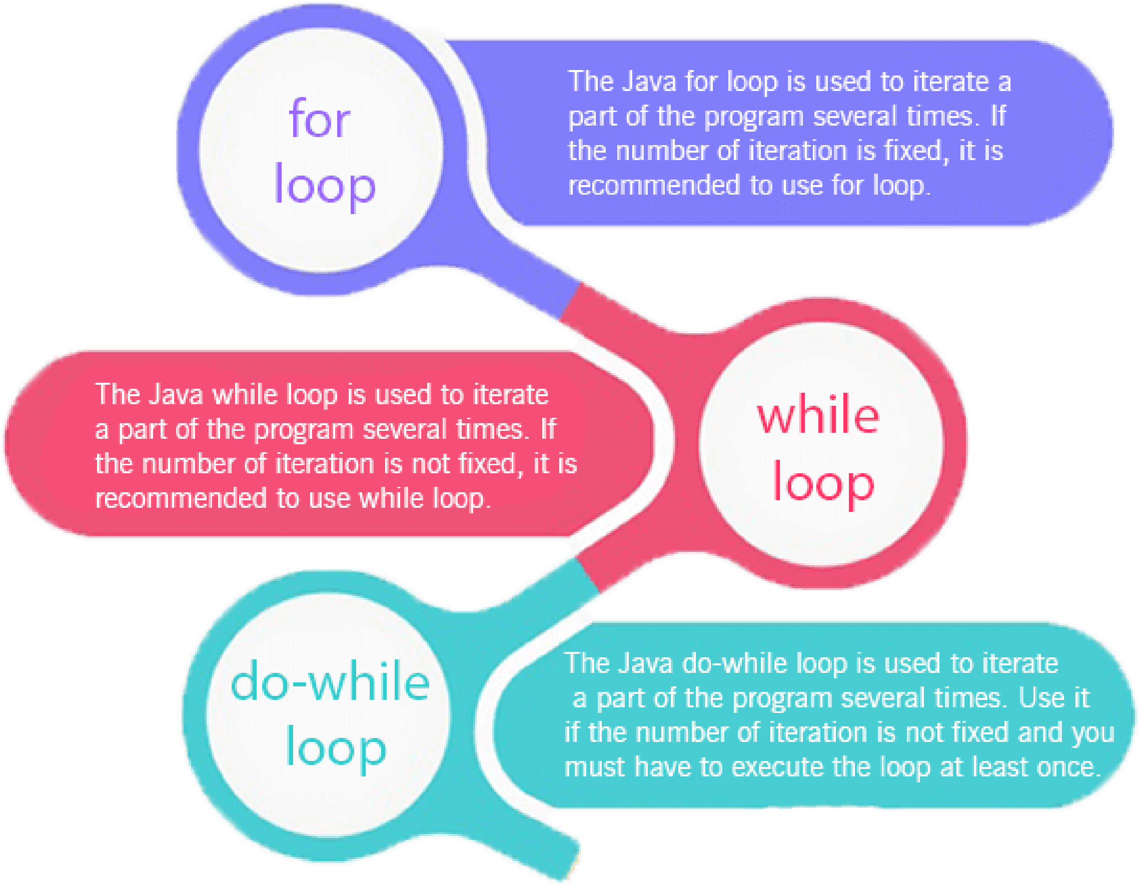
For / While Loops

- A repetition statement allows you to specify that a program should repeat an action.

- Loops can execute a block of code as long as a specified condition is reached.

- Loops are handy because they save time, reduce errors, and they make code more readable.

For / While Loops



for loop

The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

while loop

The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

do-while loop

The Java do-while loop is used to iterate a part of the program several times. Use it if the number of iteration is not fixed and you must have to execute the loop at least once.

Reference: https://www.javatpoint.com/java-for-loop

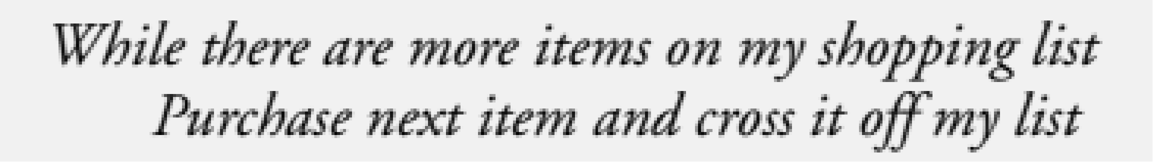
For / While Loops

- Loops Requirements:

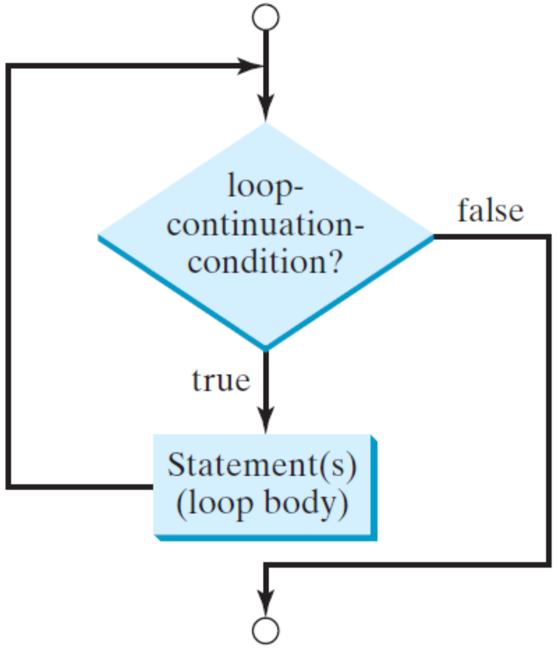
1. **A** control variable (or loop counter)
2. **T**he initial value of the control variable
3. **T**he increment by which the control variable is modified each time through the loop (also known as each iteration of the loop)
4. **T**he loop-continuation condition that determines if looping should continue.

For / While Loops

- **while Repetition Statement**
  - The pseudocode statement

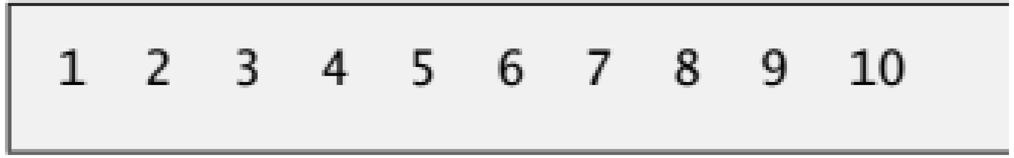


```
While there are more items on my shopping list
    Purchase next item and cross it off my list
```



For / While Loops

```
int counter = 1;                    // declare and initialize control variable
while (counter <= 10 )  {   // loop-continuation condition
        System.out.printf("%d ", counter);
        ++counter;              // increment control variable
}
System.out.println();
```

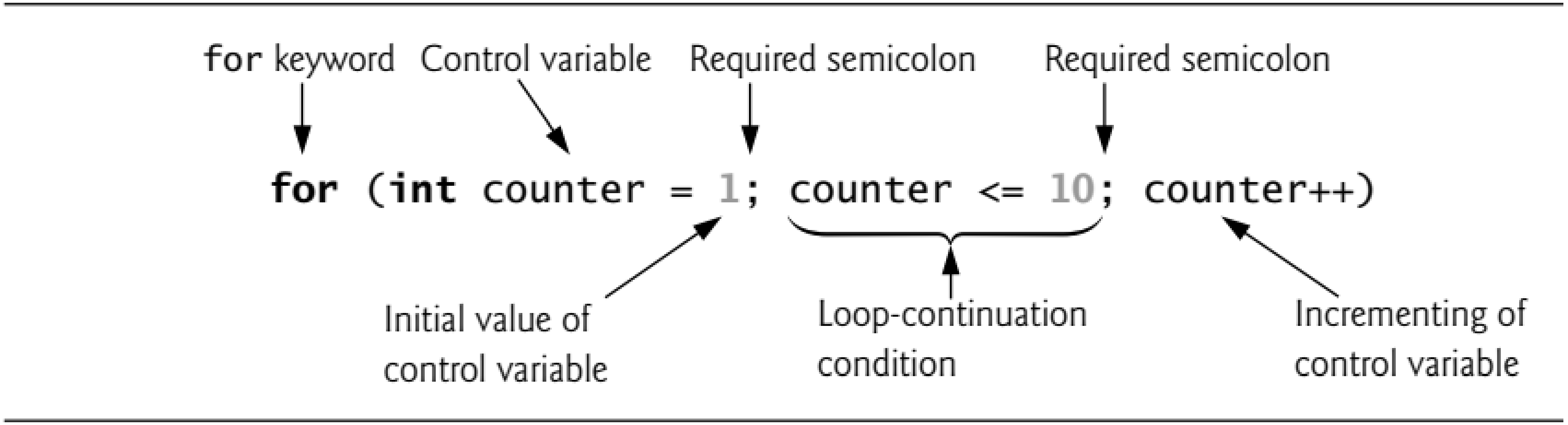


```
1  2  3  4  5  6  7  8  9  10
```

For / While Loops

- **for Repetition Statement**
  - specifies the counter-controlled-repetition details in a single line of code.

```
for (initialization; loopContinuationCondition; increment)
   statement
```



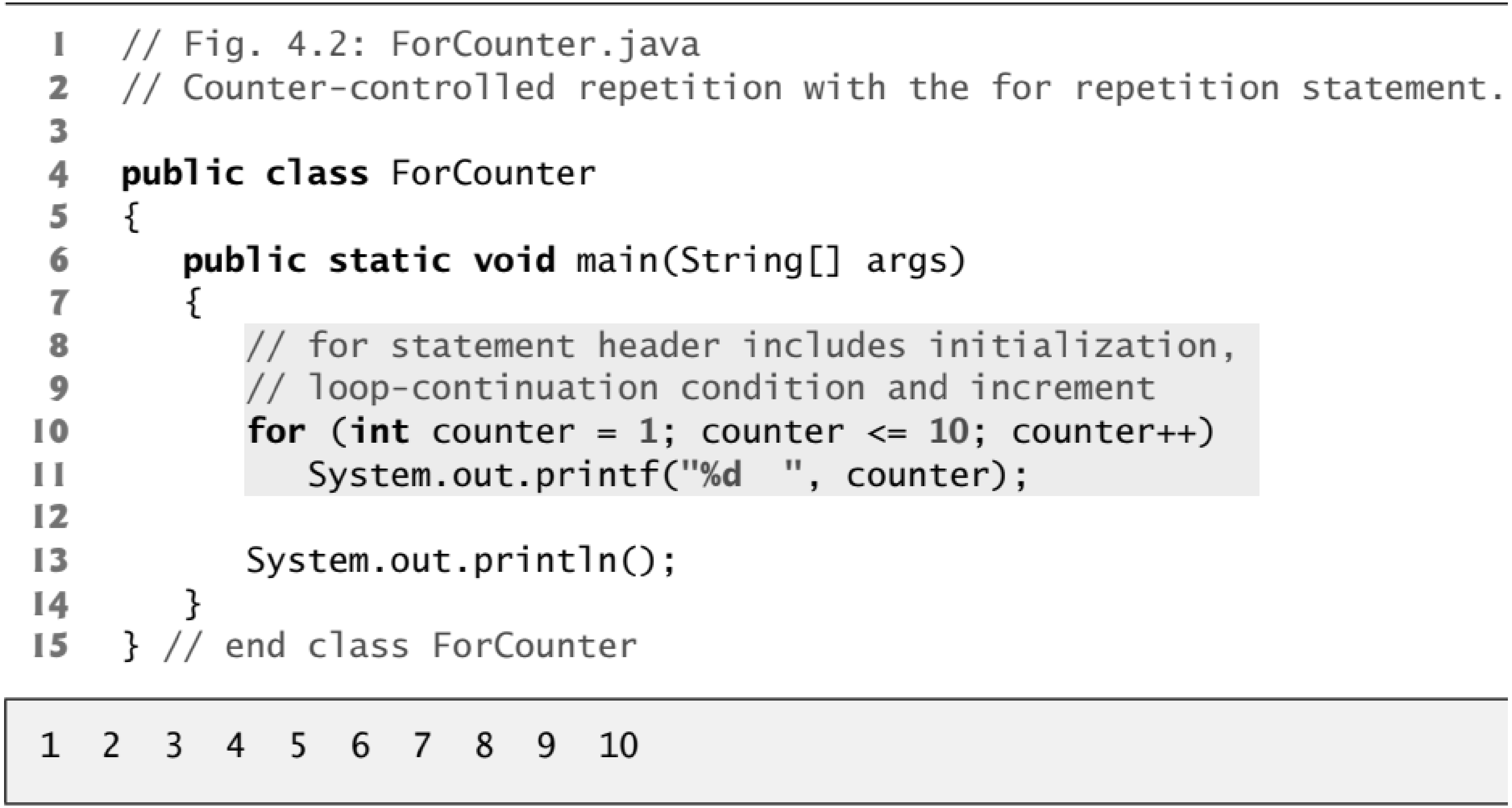
for keyword
Control variable
Required semicolon
Required semicolon
Initial value of control variable
Loop-continuation condition
Incrementing of control variable

```
for (int counter = 1; counter <= 10; counter++)
```

Fig. 4.3 | for statement header components.

For / While Loops

- for Repetition Statement



```
// Fig. 4.2: ForCounter.java
// Counter-controlled repetition with the for repetition statement.

public class ForCounter
{
   public static void main(String[] args)
   {
      // for statement header includes initialization,
      // loop-continuation condition and increment
      for (int counter = 1; counter <= 10; counter++)
         System.out.printf("%d  ", counter);

      System.out.println();
   }
} // end class ForCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.2 | Counter-controlled repetition with the `for` repetition statement.

For / While Loops

- **for Repetition Statement**

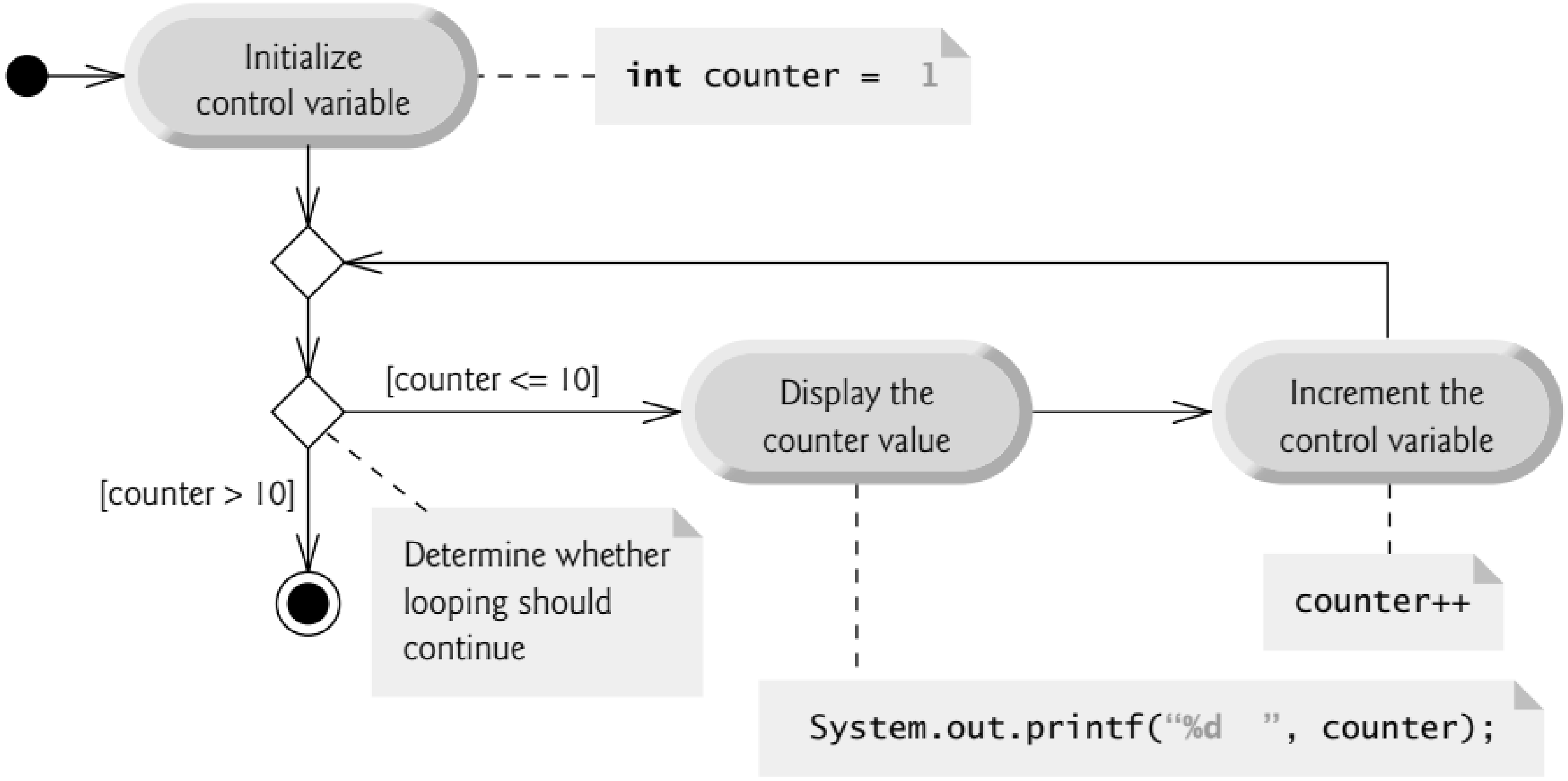
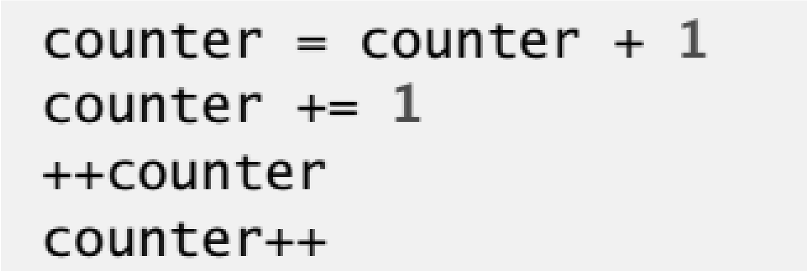


Fig. 4.4 | UML activity diagram for the `for` statement in Fig. 4.2.

For / While Loops

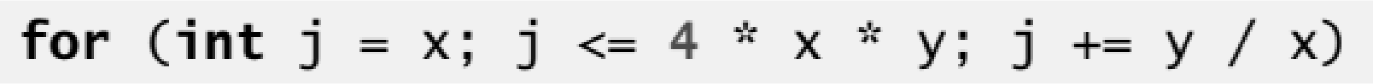
- **for Repetition Statement**
  - The increment expression in a for acts as if it were a standalone statement at the end of the for’s body. Therefore, the expressions.



```
counter = counter + 1
counter += 1
++counter
counter++
```

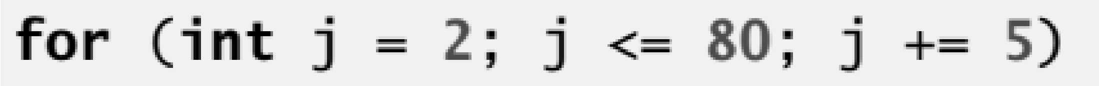
For / While Loops

- **for Repetition Statement**
  - *Placing Arithmetic Expressions in a **for** Statement's Header*
    - For example, assume that x = 2 and y = 10. If x and y are not modified in the body of the loop, the statement



```
for (int j = x; j <= 4 * x * y; j += y / x)
```

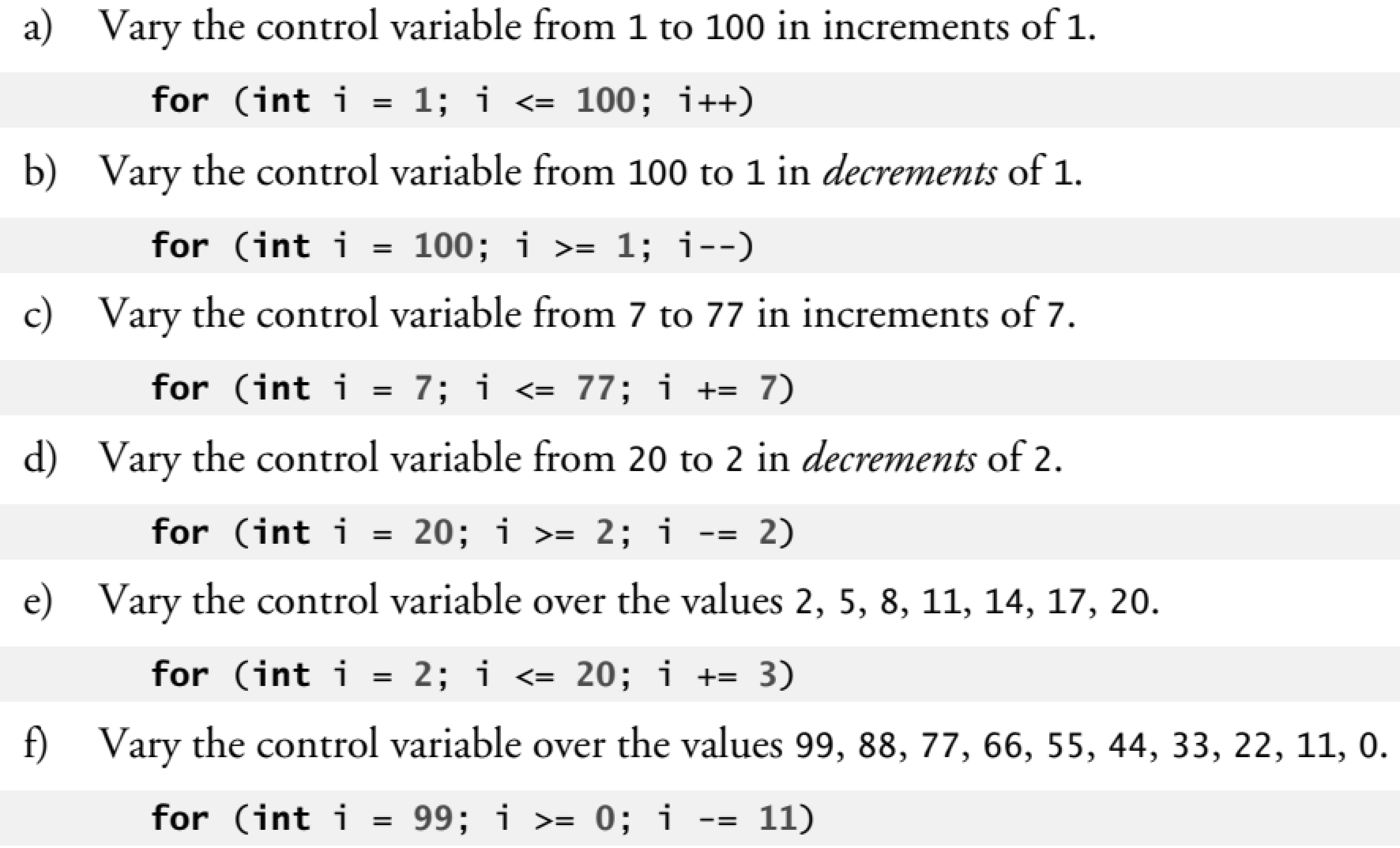
    - is equivalent to the statement



```
for (int j = 2; j <= 80; j += 5)
```

For / While Loops

- **Examples Using the for Statement**



a) Vary the control variable from 1 to 100 in increments of 1.

```
for (int i = 1; i <= 100; i++)
```

b) Vary the control variable from 100 to 1 in *decrements* of 1.

```
for (int i = 100; i >= 1; i--)
```

c) Vary the control variable from 7 to 77 in increments of 7.

```
for (int i = 7; i <= 77; i += 7)
```

d) Vary the control variable from 20 to 2 in *decrements* of 2.

```
for (int i = 20; i >= 2; i -= 2)
```

e) Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

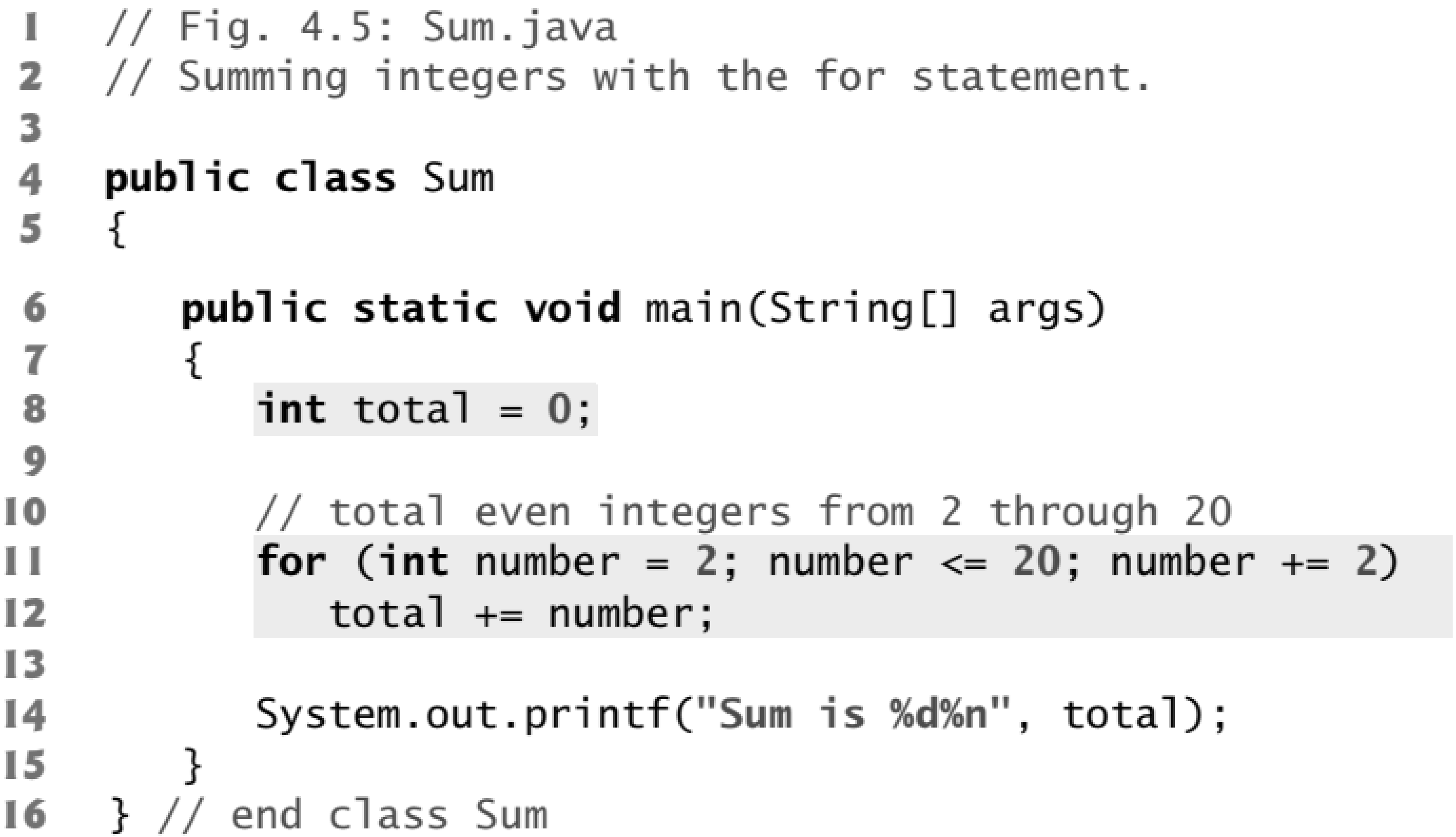
```
for (int i = 2; i <= 20; i += 3)
```

f) Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for (int i = 99; i >= 0; i -= 11)
```

For / While Loops

- *Summing the Even Integers from 2 to 20*



```
// Fig. 4.5: Sum.java
// Summing integers with the for statement.

public class Sum
{
   public static void main(String[] args)
   {
      int total = 0;

      // total even integers from 2 through 20
      for (int number = 2; number <= 20; number += 2)
         total += number;

      System.out.printf("Sum is %d%n", total);
   }
} // end class Sum
```

Output

```
Sum is 110
```

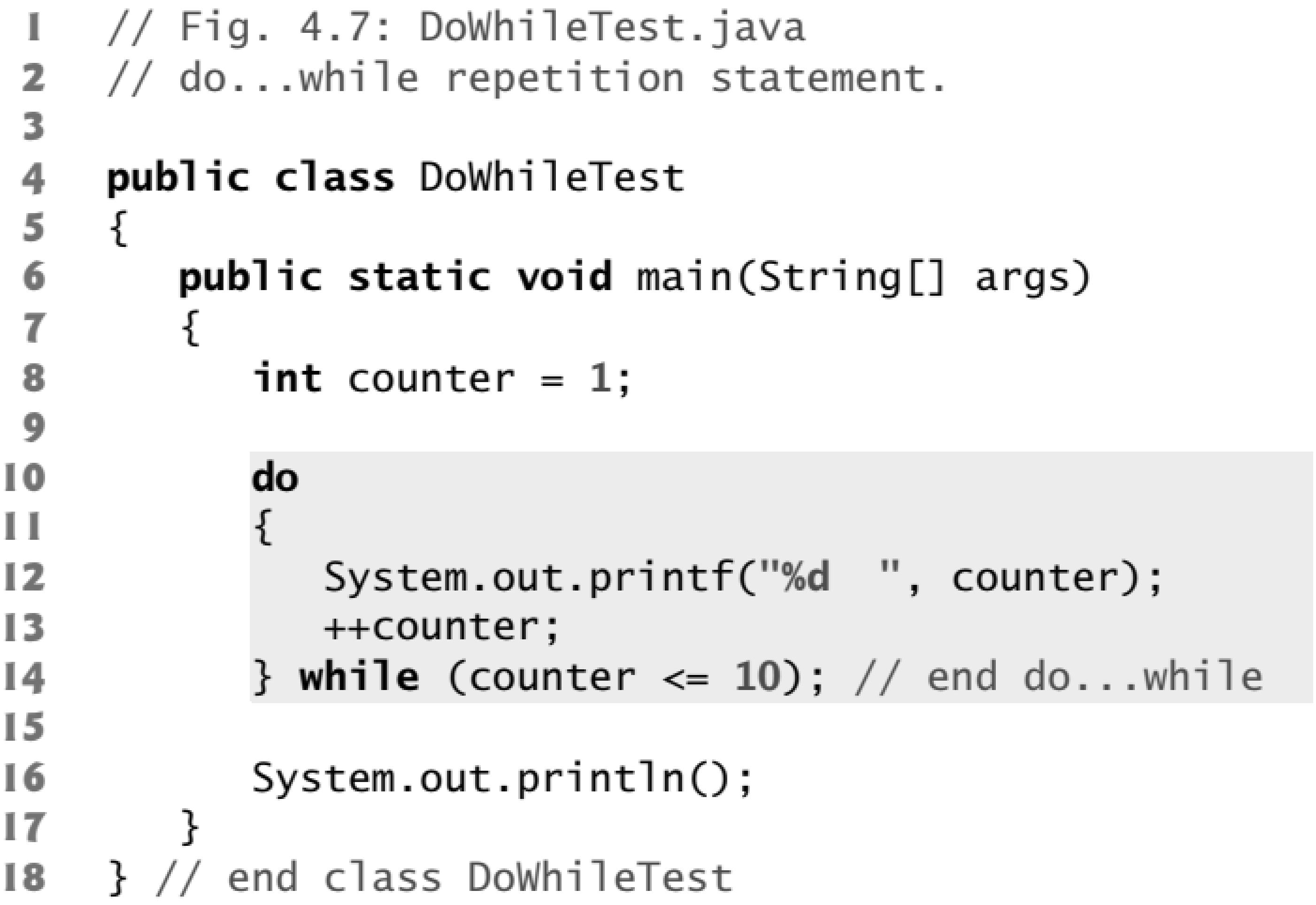
For / While Loops

- **do…while Repetition Statement**
- In the while, the program tests the loop-continuation condition
  - at the *beginning* of the loop, *before* executing the loop’s body;
  - if the condition is *false*, the body *never* executes.

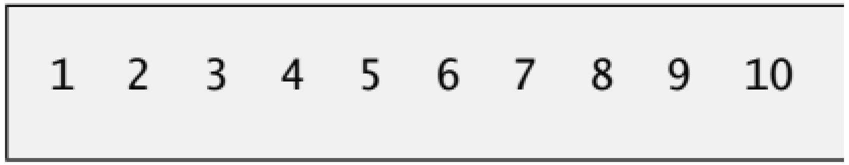
- The do…while statement tests the loop continuation condition
  - *after* executing the loop’s body;
  - therefore, *the body always executes at least once*.

For / While Loops

- do…while **Repetition Statement**



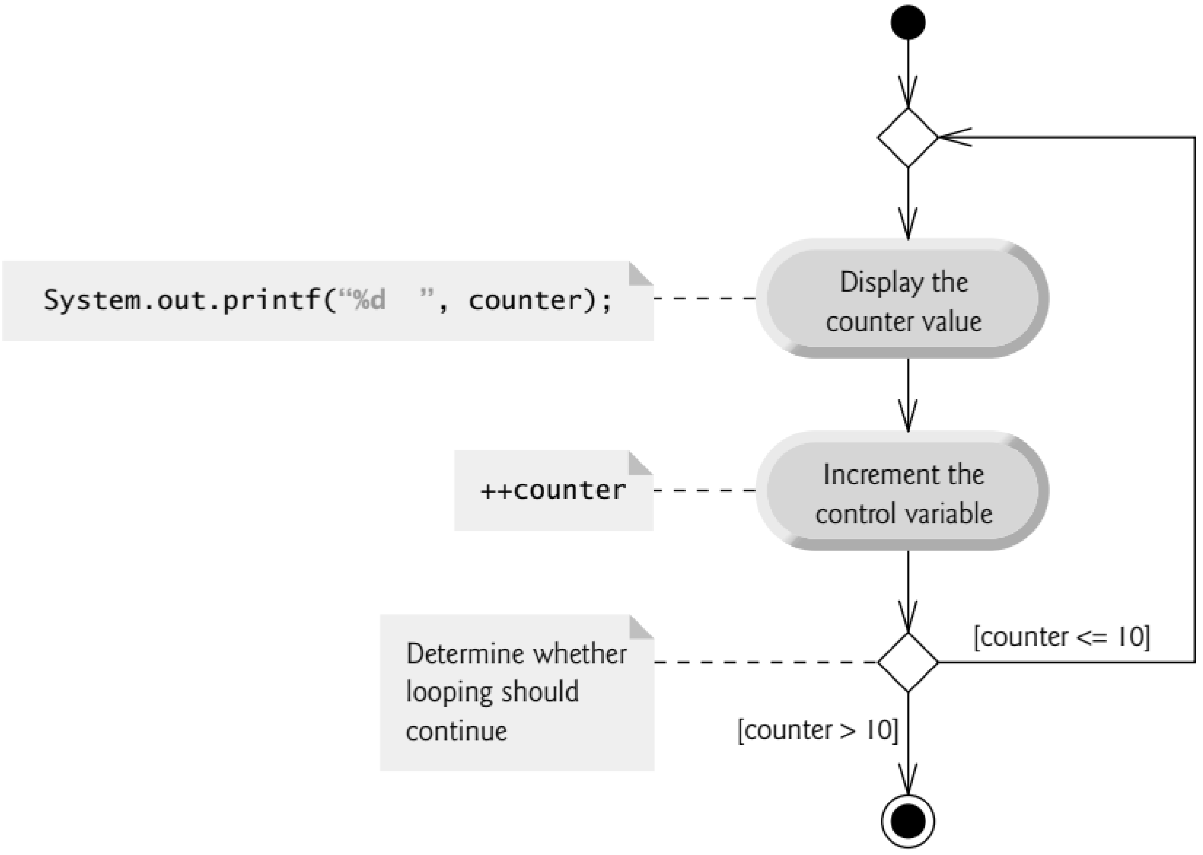
```
// Fig. 4.7: DoWhileTest.java
// do...while repetition statement.

public class DoWhileTest
{
   public static void main(String[] args)
   {
      int counter = 1;

      do
      {
         System.out.printf("%d  ", counter);
         ++counter;
      } while (counter <= 10); // end do...while

      System.out.println();
   }
} // end class DoWhileTest
```



```
1  2  3  4  5  6  7  8  9  10
```

For / While Loops

- do…while Repetition Statement

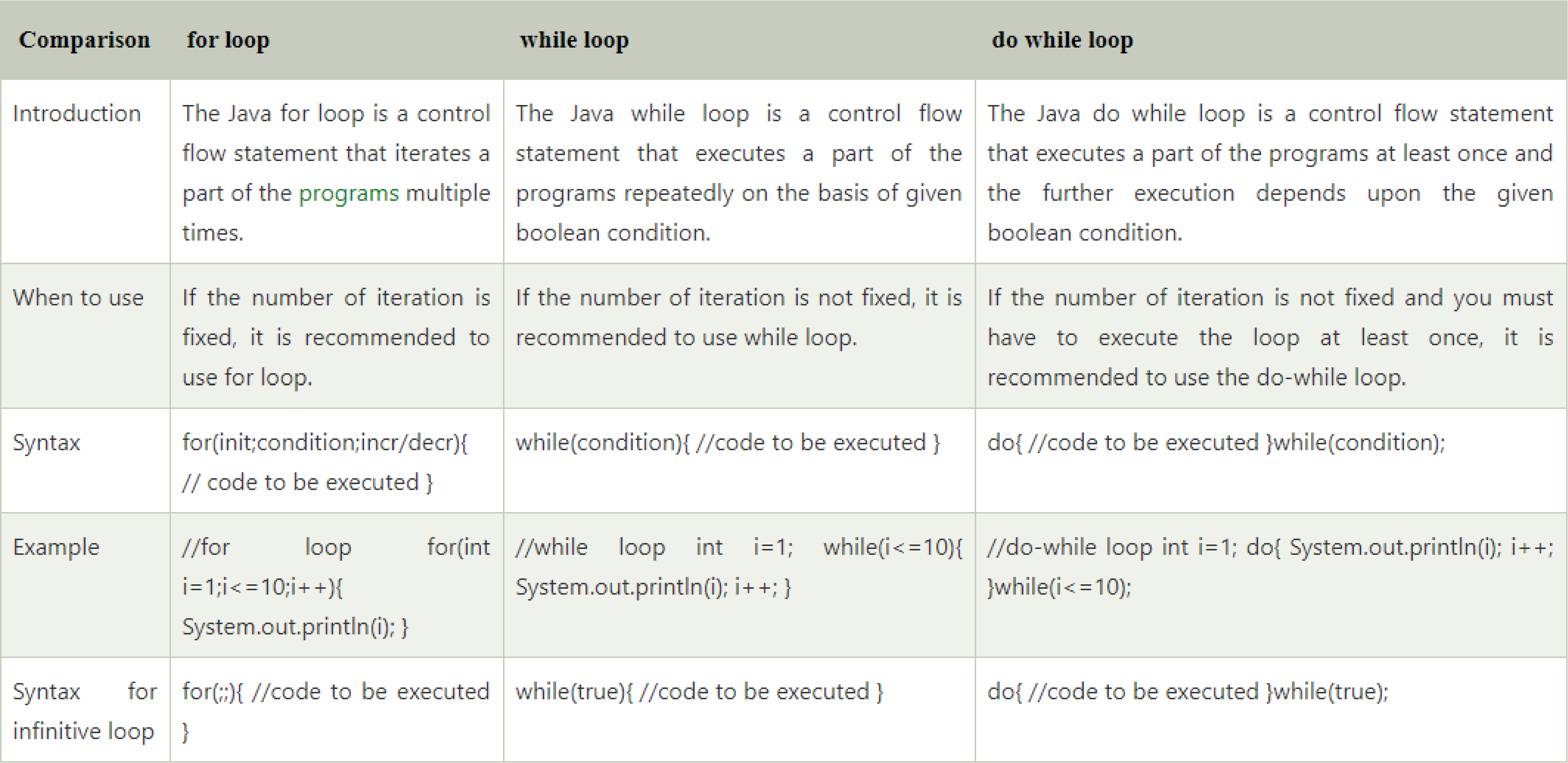


System.out.printf("%d ", counter);
Display the counter value
++counter
Increment the control variable
Determine whether looping should continue
[counter <= 10]
[counter > 10]

Fig. 4.8 | do…`while` repetition statement UML activity diagram.

For / While Loops

- ## Comparison



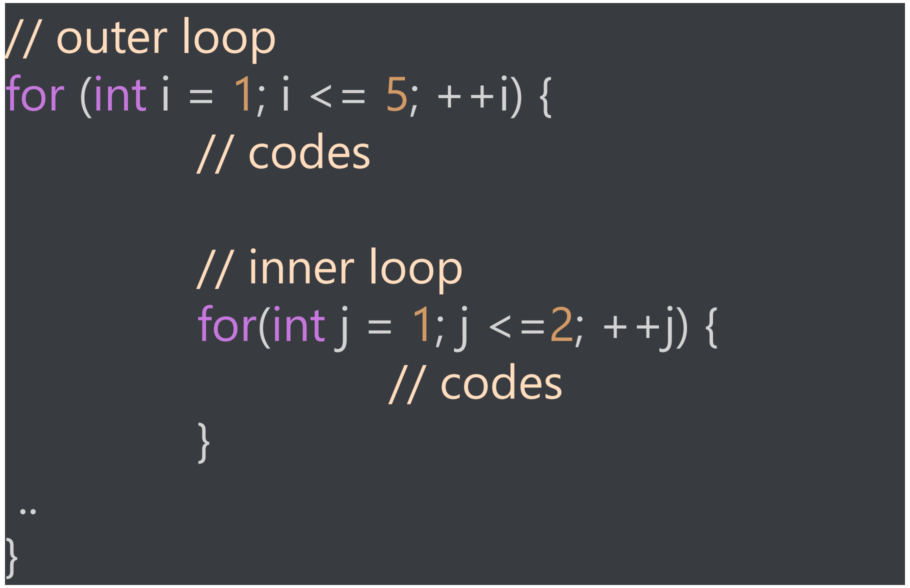
Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.
Syntax	for(init;condition;incr/decr){ // code to be executed }	while(condition){ //code to be executed }	do{ //code to be executed }while(condition);
Example	//for loop for(int i=1;i<=10;i++){ System.out.println(i); }	//while loop int i=1; while(i<=10){ System.out.println(i); i++; }	//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);
Syntax for infinitive loop	for(;;){ //code to be executed }	while(true){ //code to be executed }	do{ //code to be executed }while(true);

Reference: https://www.javatpoint.com/java-for-loop

- Nested Loops

Nested Loops

- If a loop exists inside the body of another loop, it's called a nested loop.

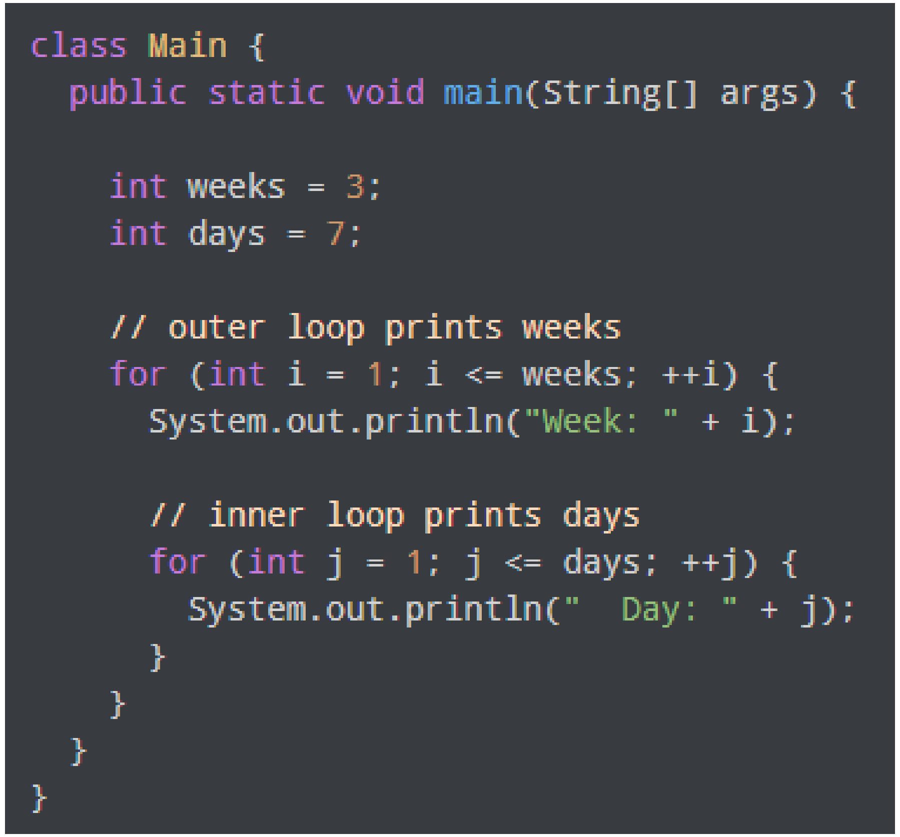


```
// outer loop
for (int i = 1; i <= 5; ++i) {
        // codes

        // inner loop
        for(int j = 1; j <=2; ++j) {
                // codes
        }
 ..
}
```

Reference: https://www.programiz.com/java-programming/nested-loop

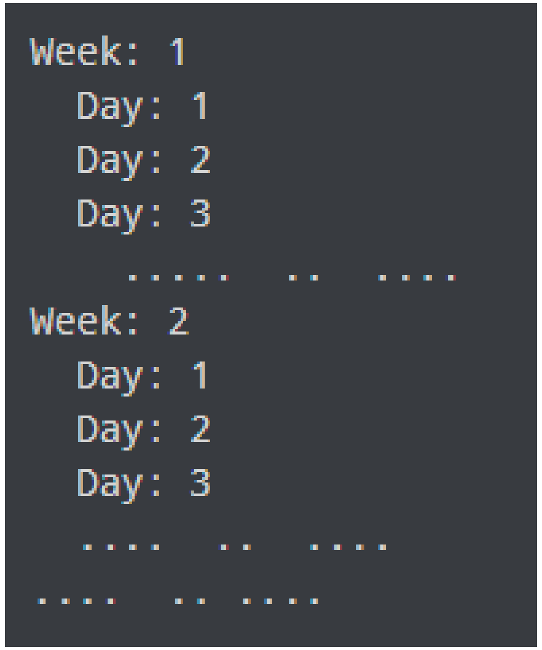
Nested Loops

- Example 1:



```
class Main {
  public static void main(String[] args) {

    int weeks = 3;
    int days = 7;

    // outer loop prints weeks
    for (int i = 1; i <= weeks; ++i) {
      System.out.println("Week: " + i);

      // inner loop prints days
      for (int j = 1; j <= days; ++j) {
        System.out.println("  Day: " + j);
      }
    }
  }
}
```

Output

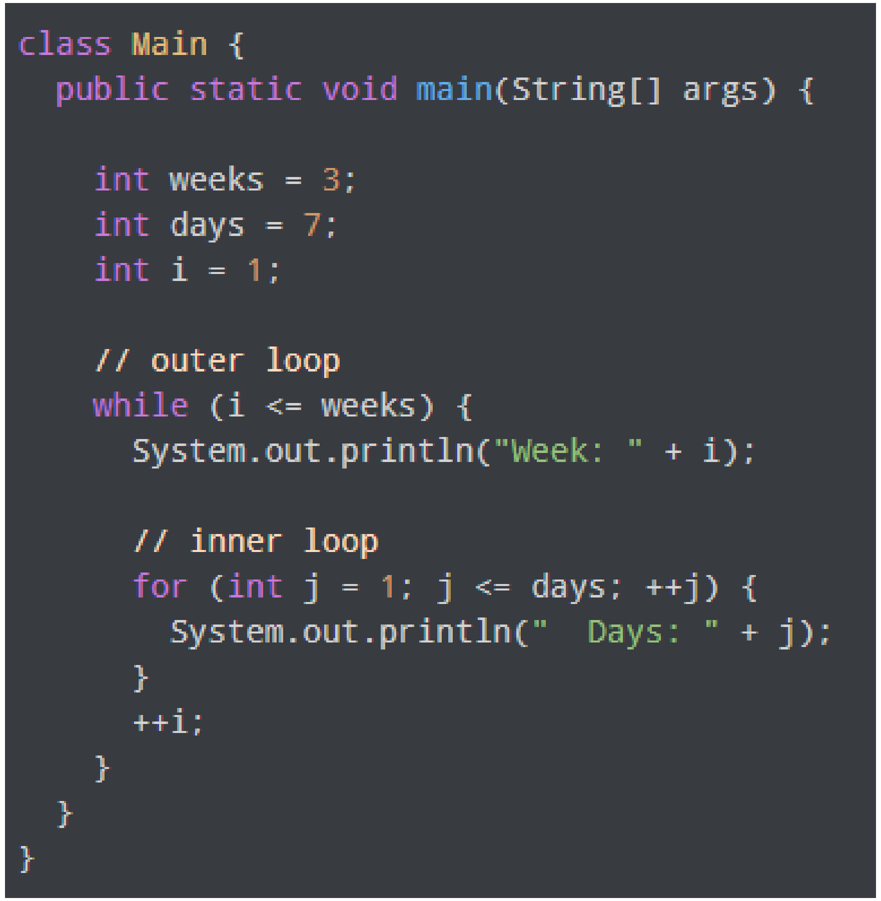


```
Week: 1
  Day: 1
  Day: 2
  Day: 3
    .....  ..  ....
Week: 2
  Day: 1
  Day: 2
  Day: 3
  ....  ..  ....
....  .. ....
```

Reference: https://www.programiz.com/java-programming/nested-loop

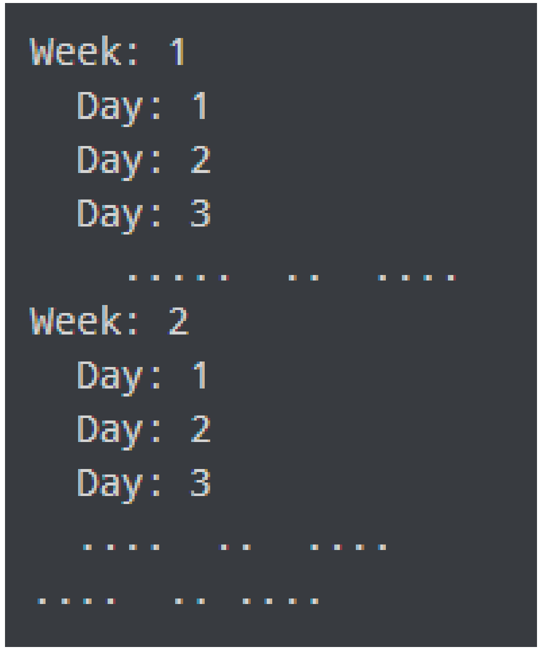
Nested Loops

- Example 2:



```
class Main {
  public static void main(String[] args) {

    int weeks = 3;
    int days = 7;
    int i = 1;

    // outer loop
    while (i <= weeks) {
      System.out.println("Week: " + i);

      // inner loop
      for (int j = 1; j <= days; ++j) {
        System.out.println("  Days: " + j);
      }
      ++i;
    }
  }
}
```

Output

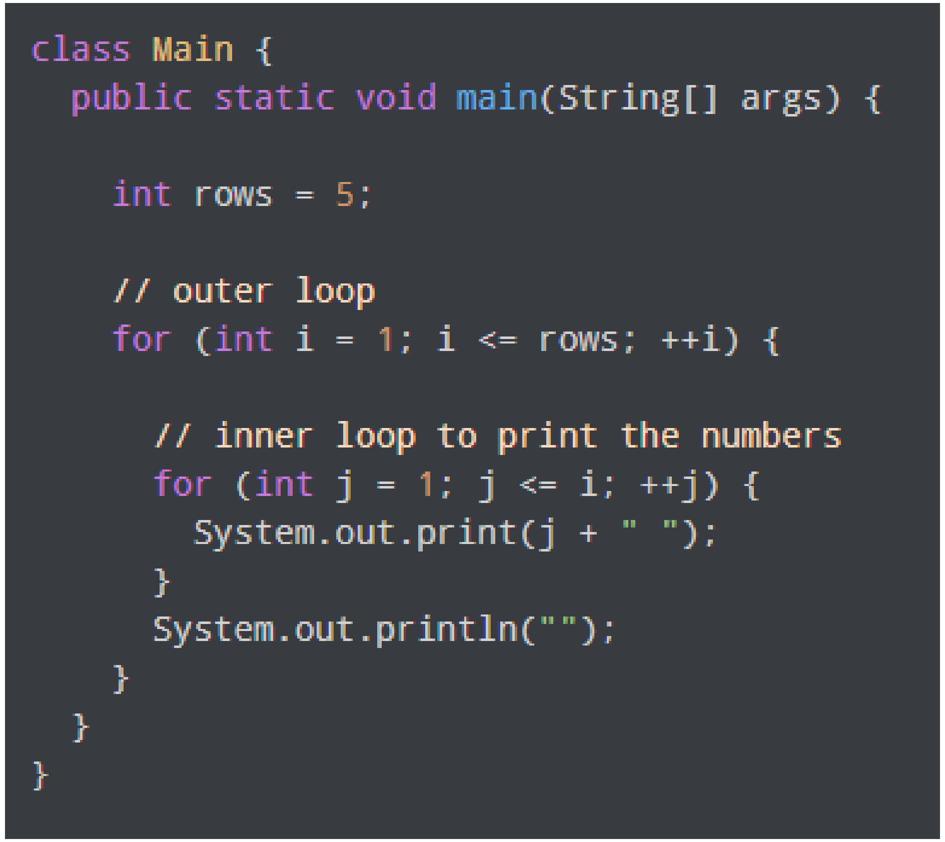


```
Week: 1
  Day: 1
  Day: 2
  Day: 3
    ......  ..  ....
Week: 2
  Day: 1
  Day: 2
  Day: 3
  ....  ..  ....
....  .. ....
```

Reference: https://www.programiz.com/java-programming/nested-loop

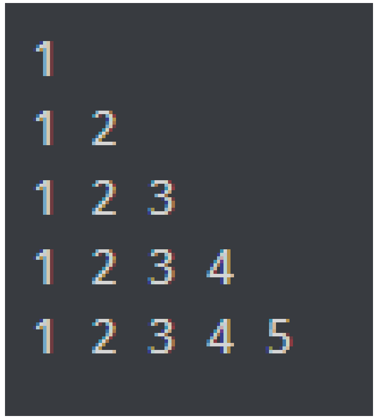
Nested Loops

- Example 3:



```
class Main {
  public static void main(String[] args) {

    int rows = 5;

    // outer loop
    for (int i = 1; i <= rows; ++i) {

      // inner loop to print the numbers
      for (int j = 1; j <= i; ++j) {
        System.out.print(j + " ");
      }
      System.out.println("");
    }
  }
}
```

Output



```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Reference: https://www.programiz.com/java-programming/nested-loop

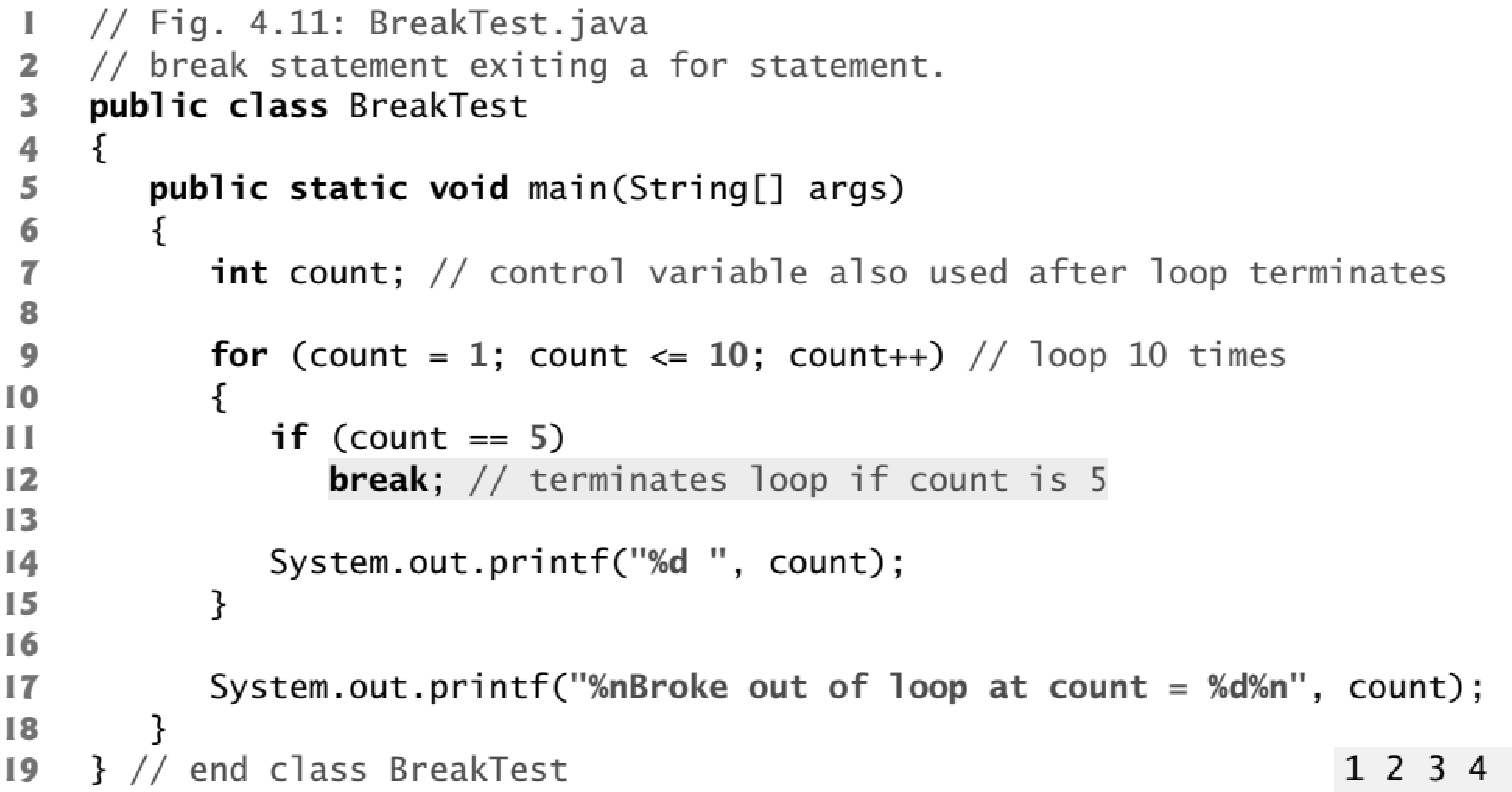
- Break and Continue Statements

Break and Continue Statements

- ***break*** *Statement*
  - When executed in a while, for, do…while or switch,
    - causes *immediate* exit from that statement.
    - Execution continues with the first statement after the control statement.
  - Common uses of the break statement are
    - to escape early from a loop
    - to skip the remainder of a switch

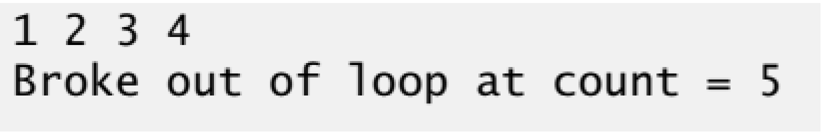
Break and Continue Statements

- ***break** Statement*



```java
// Fig. 4.11: BreakTest.java
// break statement exiting a for statement.
public class BreakTest
{
   public static void main(String[] args)
   {
      int count; // control variable also used after loop terminates

      for (count = 1; count <= 10; count++) // loop 10 times
      {
         if (count == 5)
            break; // terminates loop if count is 5

         System.out.printf("%d ", count);
      }

      System.out.printf("%nBroke out of loop at count = %d%n", count);
   }
} // end class BreakTest
```

Output



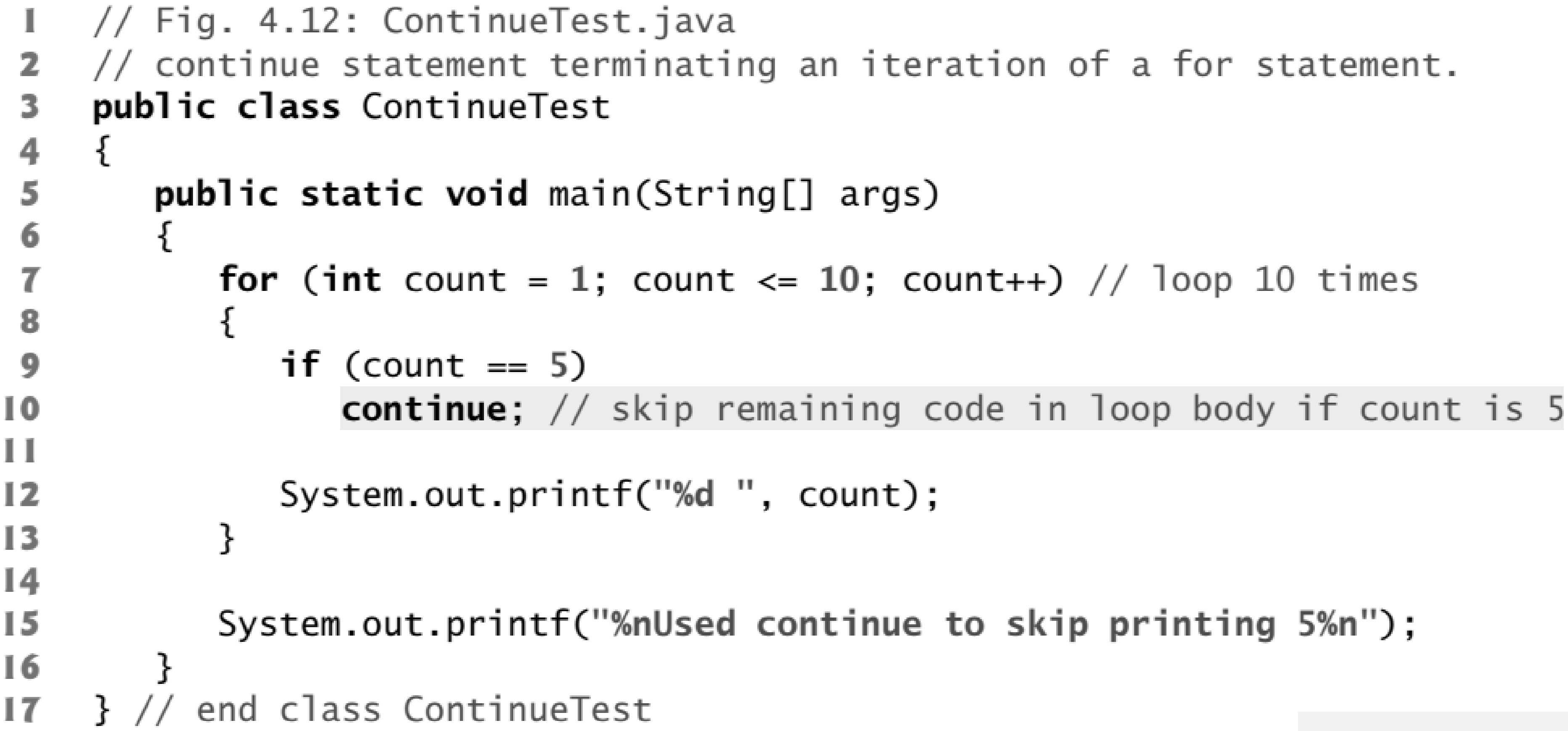
```
1 2 3 4
Broke out of loop at count = 5
```

Break and Continue Statements

- ***continue*** *Statement*
  - when executed in a while, for or do…while,
    - skips the remaining statements in the loop body
    - and proceeds with the *next iteration* of the loop.
    - In while and do…while statements, the program evaluates the loop continuation test immediately after the continue statement executes.
    - In a for statement, the increment expression executes, then the program evaluates the loop continuation test.

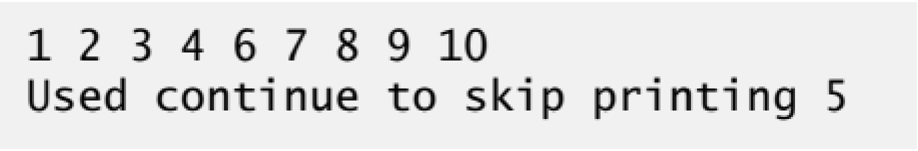
Break and Continue Statements

- ***continue** Statement*



```
// Fig. 4.12: ContinueTest.java
// continue statement terminating an iteration of a for statement.
public class ContinueTest
{
   public static void main(String[] args)
   {
      for (int count = 1; count <= 10; count++) // loop 10 times
      {
         if (count == 5)
            continue; // skip remaining code in loop body if count is 5

         System.out.printf("%d ", count);
      }

      System.out.printf("%nUsed continue to skip printing 5%n");
   }
} // end class ContinueTest
```

Output



```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

- Logical Operator

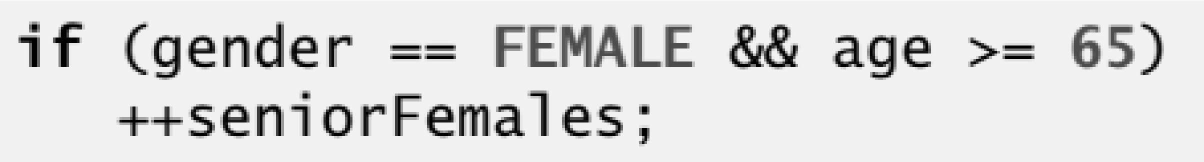
Logical Operator

- Enable you to form more complex conditions by *combining* simple conditions.

- The logical operators are
  - && (conditional AND)
  - || (conditional OR)
  - & (boolean logical AND)
  - | (boolean logical inclusive OR)
  - ^ (Boolean logical exclusive OR)
  - ! (logical NOT).

Logical Operator

- *Conditional AND (**&&**) Operator*
  - To ensure at some point in a program that two conditions are *both* true before we choose a certain path of execution.

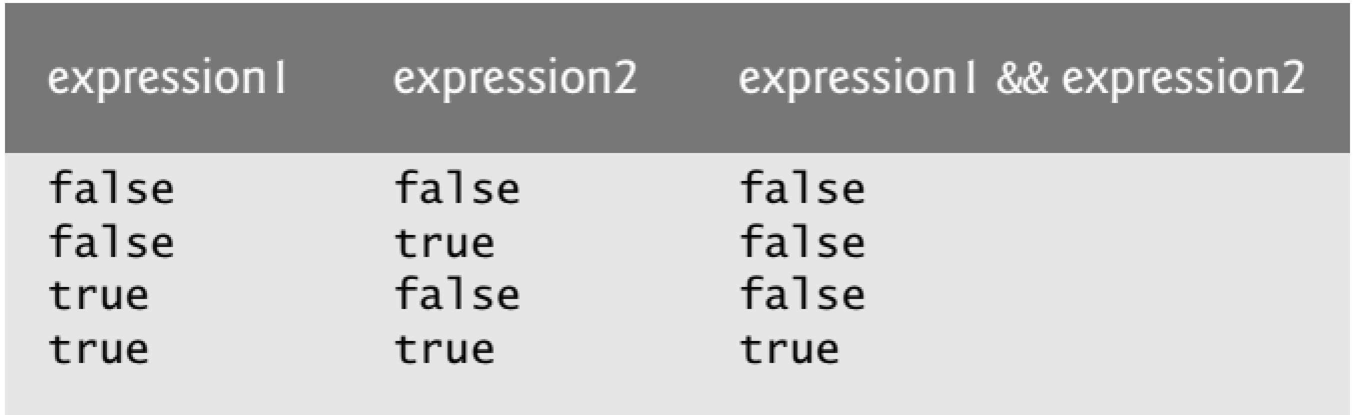


```
if (gender == FEMALE && age >= 65)
   ++seniorFemales;
```

  - The if statement condition is true if and only if *both* simple conditions are true.

Logical Operator

- *Conditional AND (**&&**) Operator*

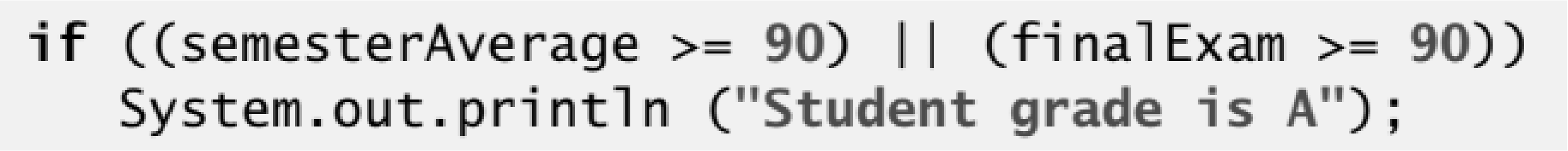


expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 4.13 | && (conditional AND) operator truth table.

Logical Operator

- *Conditional OR (||) Operator*
    - To ensure that *either or both* of two conditions are true before we choose a certain path of execution.

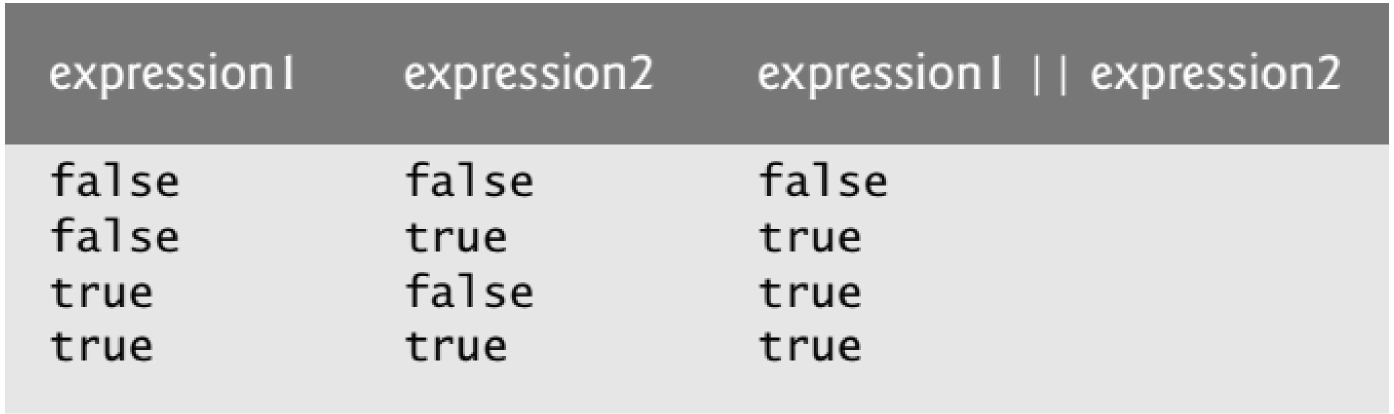


```
if ((semesterAverage >= 90) || (finalExam >= 90))
   System.out.println ("Student grade is A");
```

- The if statement condition awards the student an A if *either or both* of the simple conditions are true.

Logical Operator

- *Conditional OR (||) Operator*

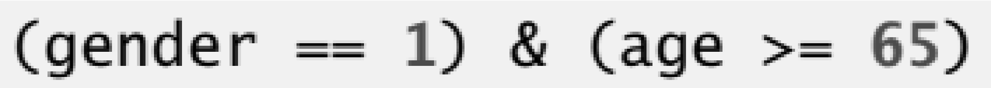


expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 4.14 | || (conditional OR) operator truth table.

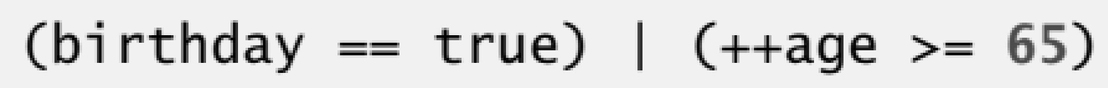
Logical Operator

- *Boolean Logical AND (&) and Boolean Logical Inclusive OR (|) Operators*
  - are identical to the && and || operators, except that the & and | operators *always* evaluate *both* of their operands.



```
(gender == 1) & (age >= 65)
```

  - evaluates age >= 65 *regardless* of whether gender is equal to 1. This is useful if the right operand has a required side effect.

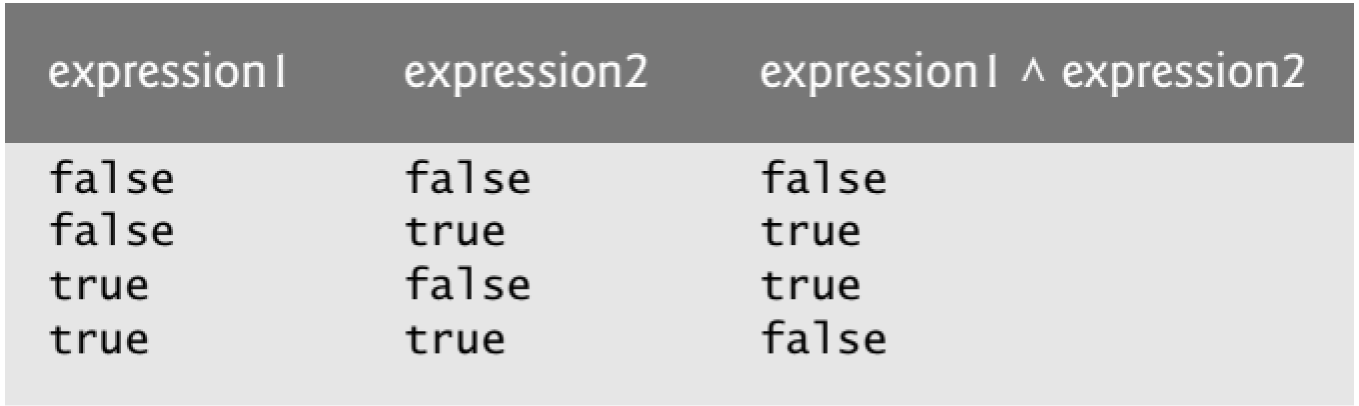


```
(birthday == true) | (++age >= 65)
```

  - guarantees that the condition ++age >= 65 will be evaluated.
    - Thus, the variable age is incremented, regardless of whether the overall expression is true or false.

Logical Operator

- *Boolean Logical Exclusive OR (^)*
    - is true *if and only if one of its operands is true and the other is false*.

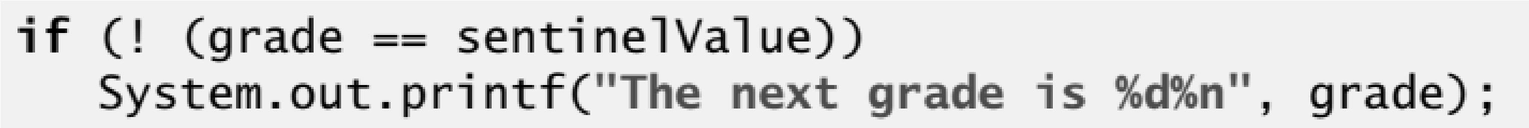


expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 4.15 | ^ (boolean logical exclusive OR) operator truth table.

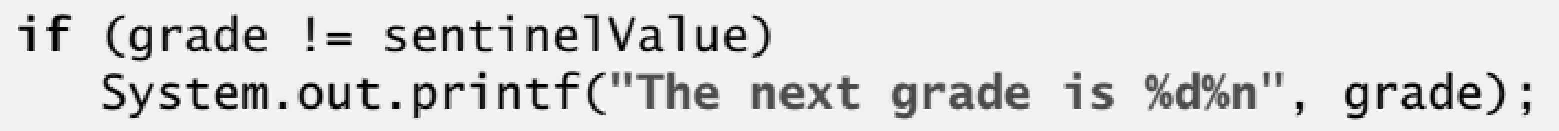
Logical Operator

- *Logical Negation (!) Operator*
  - This operator “reverses” the meaning of a condition.

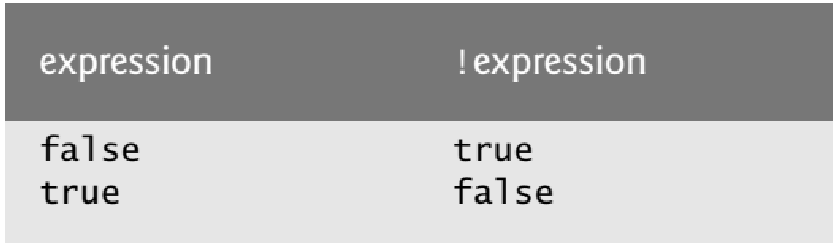


```
if (! (grade == sentinelValue))
   System.out.printf("The next grade is %d%n", grade);
```

  - which executes the printf call only if grade is *not* equal to sentinelValue.



```
if (grade != sentinelValue)
   System.out.printf("The next grade is %d%n", grade);
```

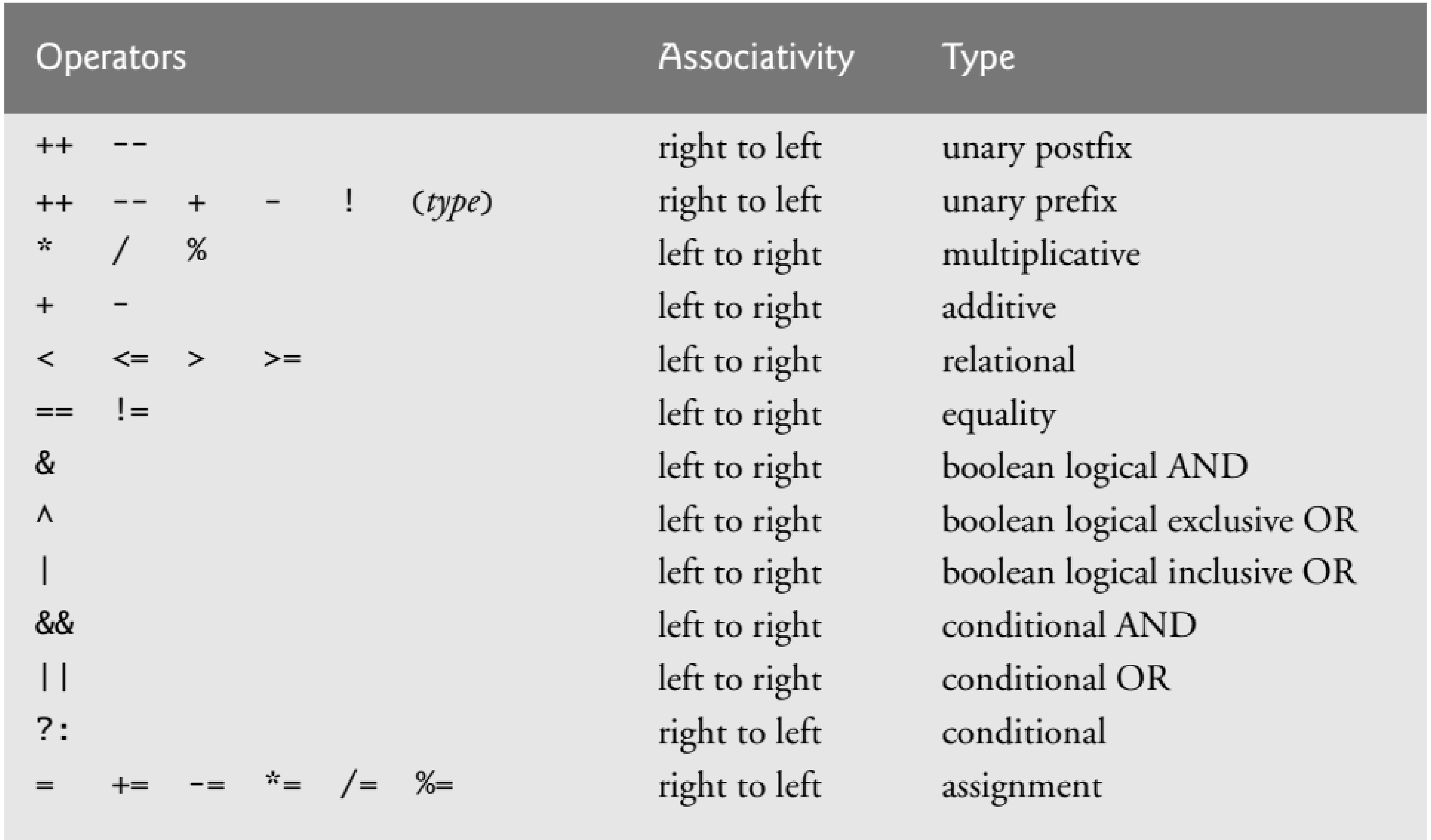


expression	!expression
`false`	`true`
`true`	`false`

Fig. 4.16 | ! (logical NOT) operator truth table.

Logical Operator

- *Precedence and Associativity of the Operators Presented So Far*



Operators	Associativity	Type
`++ --`	right to left	unary postfix
`++ -- + - !` *(type)*	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&`	left to right	boolean logical AND
`^`	left to right	boolean logical exclusive OR
`|`	left to right	boolean logical inclusive OR
`&&`	left to right	conditional AND
`||`	left to right	conditional OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 4.18 | Precedence/associativity of the operators discussed so far.

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية

SAUDI ELECTRONIC UNIVERSITY

2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230

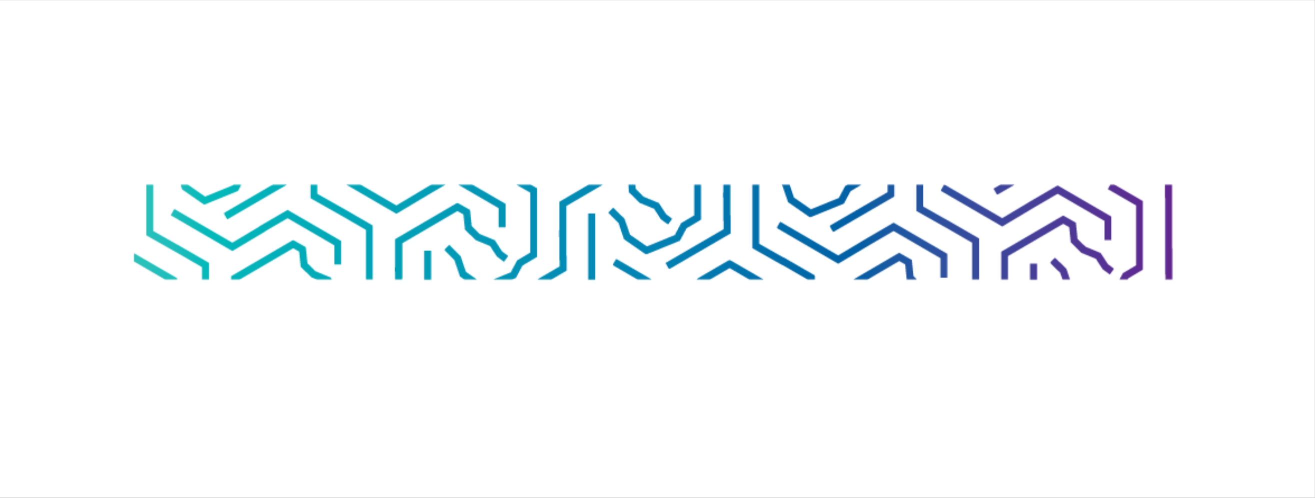
Object Oriented Programming

Week 7

Methods

Contents

1. Modules Approach
2. Declaring Methods
3. Arguments and Casting
4. Java API Packages
5. Scope of Declaration
6. Method Overloading



Weekly Learning Outcomes

- Describe Methods and calling stack in Java
- Summarize methods and Java API packages features

Required Reading

1. Chapter 5 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 6 - (Java The Complete Reference, Eleventh Edition)
2. Defining Methods:
   https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html
3. Java API Packages: https://www.javatpoint.com/package

- Modules Approach

Modules Approach

- Suppose you want to build a program as follows:

```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Modules Approach

- Defining a method will make you code more simpler

```
public static int sum(int i1, int i2) {
  int sum = 0;
  for (int i = i1; i <= i2; i++)
    sum += i;
  return sum;
}
```

```
public static void main(String[] args) {
  System.out.println("Sum from 1 to 10 is " + sum(1, 10));
  System.out.println("Sum from 20 to 30 is " + sum(20, 30));
  System.out.println("Sum from 35 to 45 is " + sum(35, 45));
}
```

Modules Approach

Java programs by combining new methods and classes with predefined ones available in the Java Application Programming Interface.

Methods help you modularize a program by separating its tasks into self-contained units.

Divide and Conquer

- It makes program development more manageable by constructing programs from small, simple pieces.
- Software reusability—using existing methods as building blocks to create new programs.

Modules Approach

- *Hierarchical Relationship Between Method Calls*
  - A method is invoked by a method call
  - when the called method completes its task
    - it returns control—and possibly a result—to the caller.

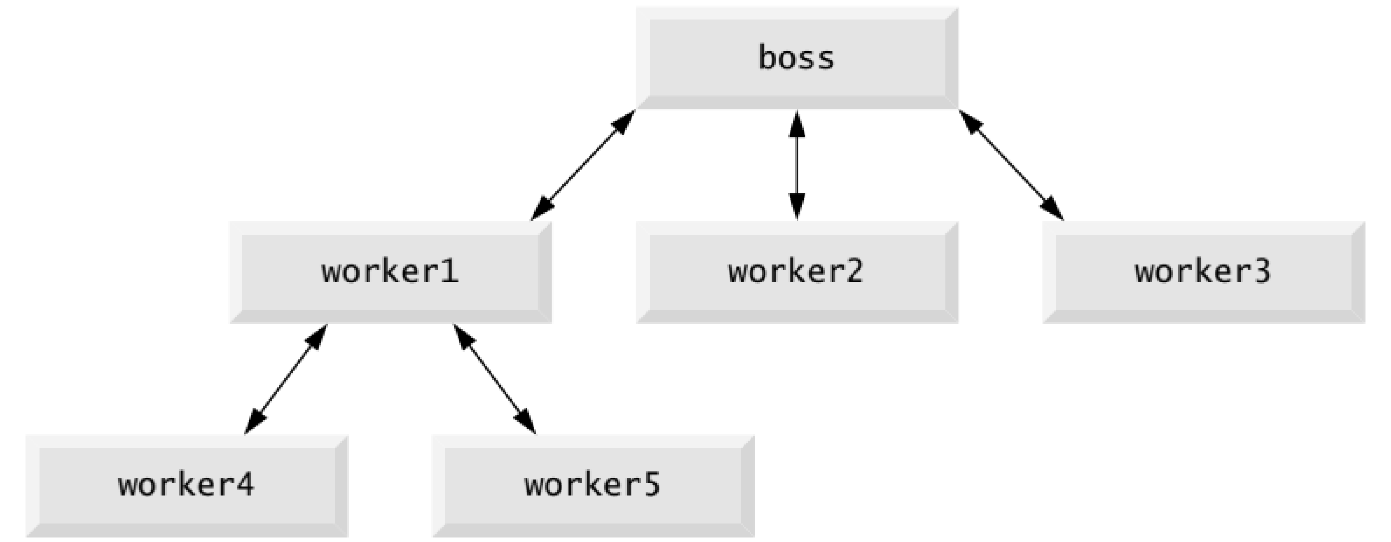


Fig. 5.1 | Hierarchical boss-method/worker-method relationship.

- Declaring Methods

Declaring Methods

```
public static int methodName(int a, int b) {
         // body
}
```

public static – modifier //it will be discussed in detail in week 9

int – return type

methodName – name of the method

a, b – formal parameters

int a, int b – list of parameters

Reference: https://www.tutorialspoint.com/java/java_methods.htm

Declaring Methods

- The method name should be a **verb** and start with a **lowercase** letter.
  - If the method name has more than two words, the first name must be a verb followed by adjective or noun.

  - In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:

  - **Single-word method name:** sum(), area()

  - **Multi-word method name:** areaOfCircle(), stringComparision()

Reference: https://www.javatpoint.com/method-in-java

Declaring Methods

- Types of Method
  - Predefined Method
  - User-defined Method

Reference: https://www.javatpoint.com/method-in-java

Declaring Methods

- Predefined Method
  - are the method that is already defined in the Java class libraries.
  - known as the **standard library method** or **built-in method**.

```
public class Demo
{
        public static void main(String[] args)
        {
                // using the max() method of Math class
                System.out.print("The maximum number is: " + Math.max(9,7));
        }
}
```

Reference: https://www.javatpoint.com/method-in-java

Declaring Methods

- User-defined Method
  - The method written by the user or programmer

```
import java.util.Scanner;
public class EvenOdd   {
public static void main (String args[])  {
            //creating Scanner class object
            Scanner scan=new Scanner(System.in);
            System.out.print("Enter the number: ");
            //reading value from the user
            int num=scan.nextInt();
            //method calling
            findEvenOdd(num);
}
```

```
//user defined method
public static void findEvenOdd(int num)
{
            //method body
            if(num%2==0)
                        System.out.println(num+" is even");
            else
                        System.out.println(num+" is odd");
}
```

Reference: https://www.javatpoint.com/method-in-java

Declaring Methods

- Example:

```
public class Addition   {
          public static void main(String[] args)   {
                    int a = 19;
                    int b = 5;
                    //method calling
                    int c = add(a, b);   //a and b are actual parameters
                    System.out.println("The sum of a and b is= " + c);
          }

          //user defined method
          public static int add(int n1, int n2)   //n1 and n2 are formal para
meters  {
                    int s;
                    s=n1+n2;
                    return s; //returning the sum
          }
}
```

Reference: https://www.javatpoint.com/method-in-java

Declaring Methods

- Return Values
    - The void keyword, used in the examples above, indicates that the method should not return a value.
    - If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

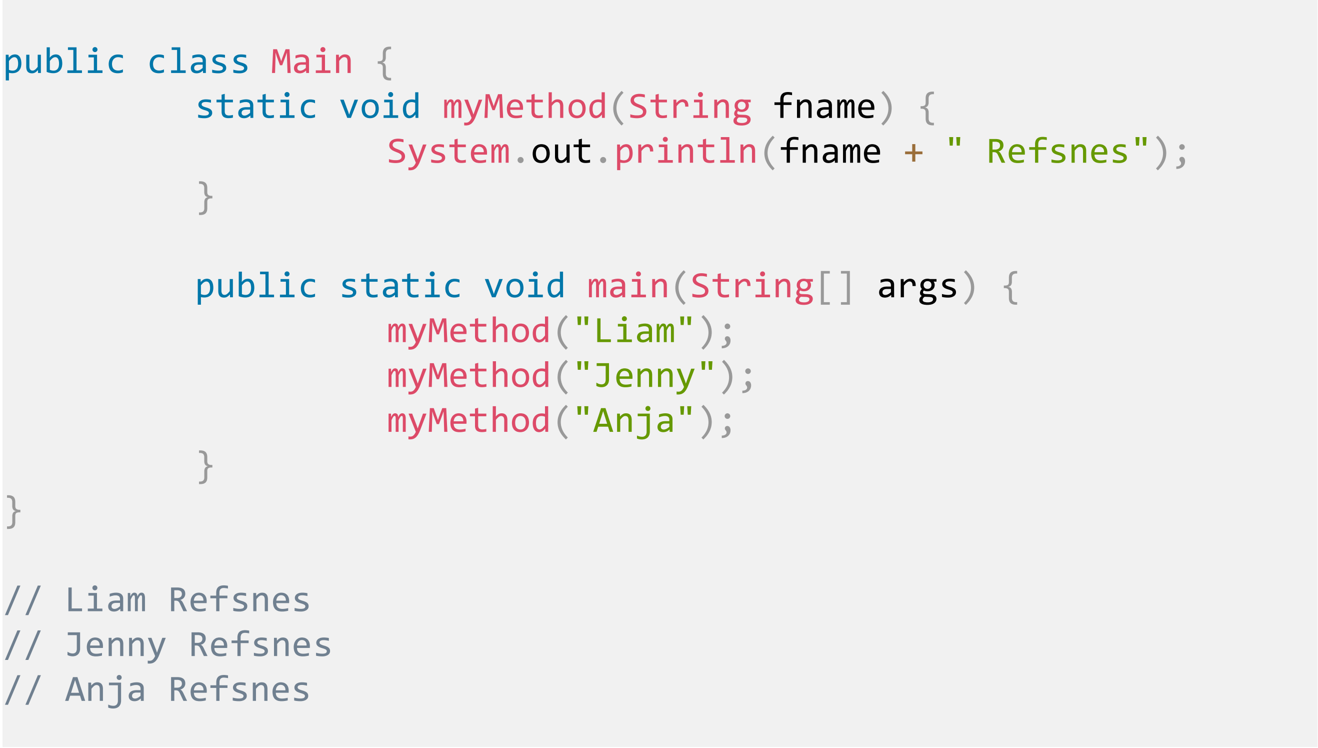
```java
public class Main {
        static int myMethod(int x) {
                return 5 + x;
        }
        public static void main(String[] args) {
                System.out.println(myMethod(3));
        }
}
// Outputs 8 (5 + 3)
```

Reference: https://www.w3schools.com/java/java_methods_param.asp

- Arguments and Casting

Arguments and Casting

- Information can be passed to methods as parameter
  - Parameters are specified after the method name, inside the parentheses.
  - You can add as many parameters as you want, just separate them with a comma.



```
public class Main {
        static void myMethod(String fname) {
                System.out.println(fname + " Refsnes");
        }

        public static void main(String[] args) {
                myMethod("Liam");
                myMethod("Jenny");
                myMethod("Anja");
        }
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Reference: https://www.w3schools.com/java/java_methods_param.asp

Arguments and Casting

- Multiple Parameters

```
public class Main {
        static void myMethod(String fname, int age) {
                System.out.println(fname + " is " + age);
        }

        public static void main(String[] args) {
                myMethod("Liam", 5);
                myMethod("Jenny", 8);
                myMethod("Anja", 31);
        }
}
// Liam is 5
// Jenny is 8
// Anja is 31
```

Reference: https://www.w3schools.com/java/java_methods_param.asp

Arguments and Casting

- **pass-by-value**
  - When Java passes a variable to a method via a parameter, the method itself receives a copy of the variable’s value, not the variable itself.

```
public class ChangeParameters {
		public static void main(String[] args) {
				int number = 1;
				tryToChangeNumber(number);
				System.out.println(number);
		}

		public static void tryToChangeNumber(int i) {
				i = 2;
		}
}
```

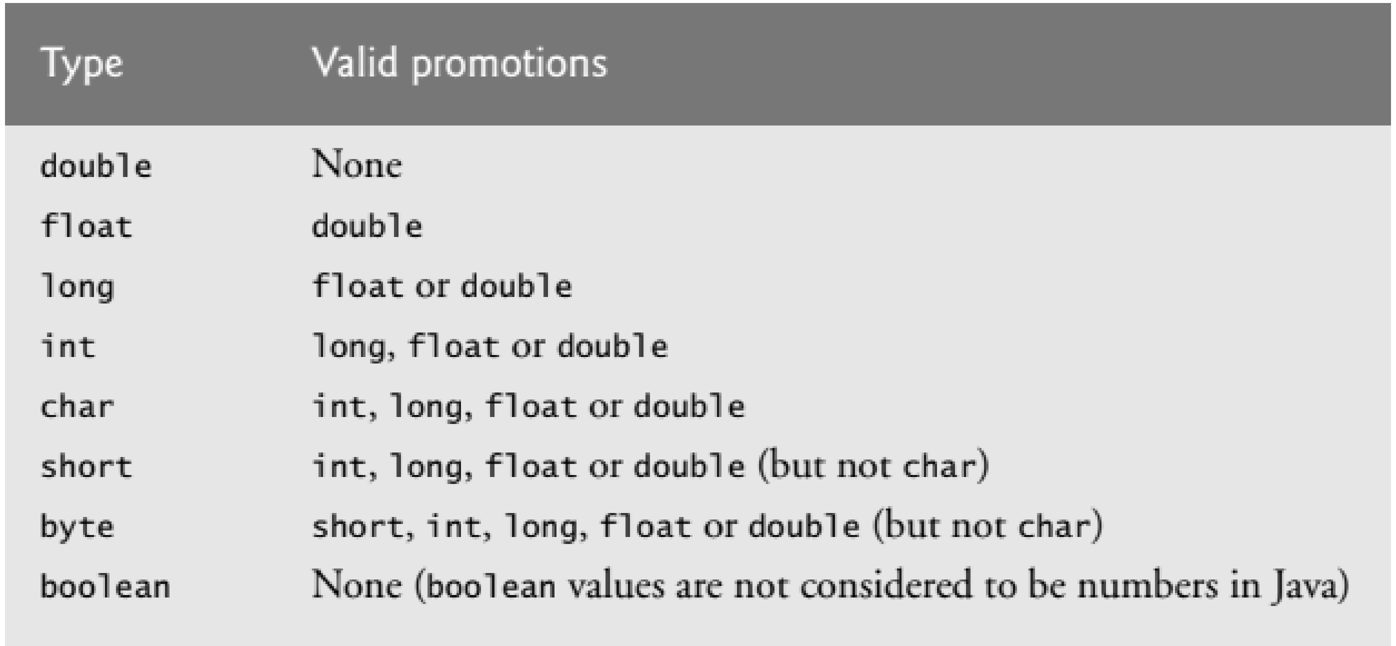
Reference: https://www.dummies.com/programming/java/how-to-use-methods-that-take-parameters-in-java/

Arguments and Casting

- **Argument Promotion**
  - Coercion of arguments
    - Forcing arguments to appropriate type to pass to method
      - e.g., `System.out.println( Math.sqrt( 4 ) );`
        - Evaluates `Math.sqrt( 4 )`
        - Then evaluates `System.out.println()`
  - Promotion rules
    - Specify how to convert types without data loss

Arguments and Casting

- **Argument Promotion**

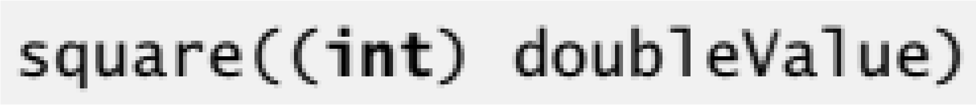


Type	Valid promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double` (but not `char`)
`byte`	`short`, `int`, `long`, `float` or `double` (but not `char`)
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Fig. 5.4 | Promotions allowed for primitive types.

Arguments and Casting

- **Argument Promotion**
  - Suppose method ***square*** calculates the square of an integer
  - Thus requires an ***int*** argument.
  - To call ***square*** with a ***double*** argument named doubleValue, we would be required to write the method call as



```
square((int) doubleValue)
```

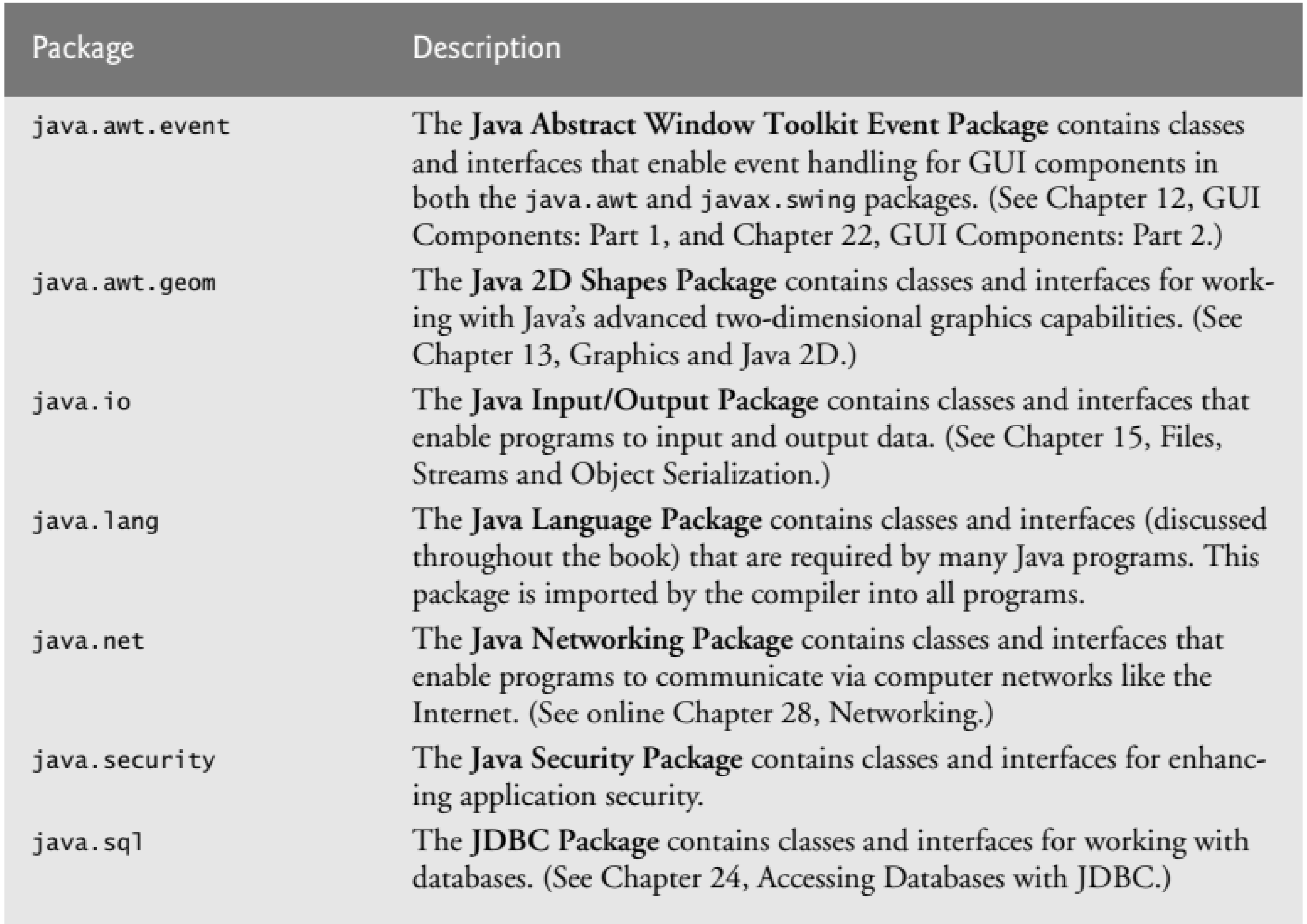
  - This method call explicitly casts (converts) doubleValue’s value to a temporary integer for use in method square.
  - Thus, if doubleValue’s value is 4.5, the method receives the value 4 and returns 16, not 20.25.

- Java API Packages

Java API Packages

- Packages
  - Classes grouped into categories of related classes
  - Promotes software reuse
  - `import` statements specify classes used in Java programs
    - e.g., `import javax.swing.JApplet;`

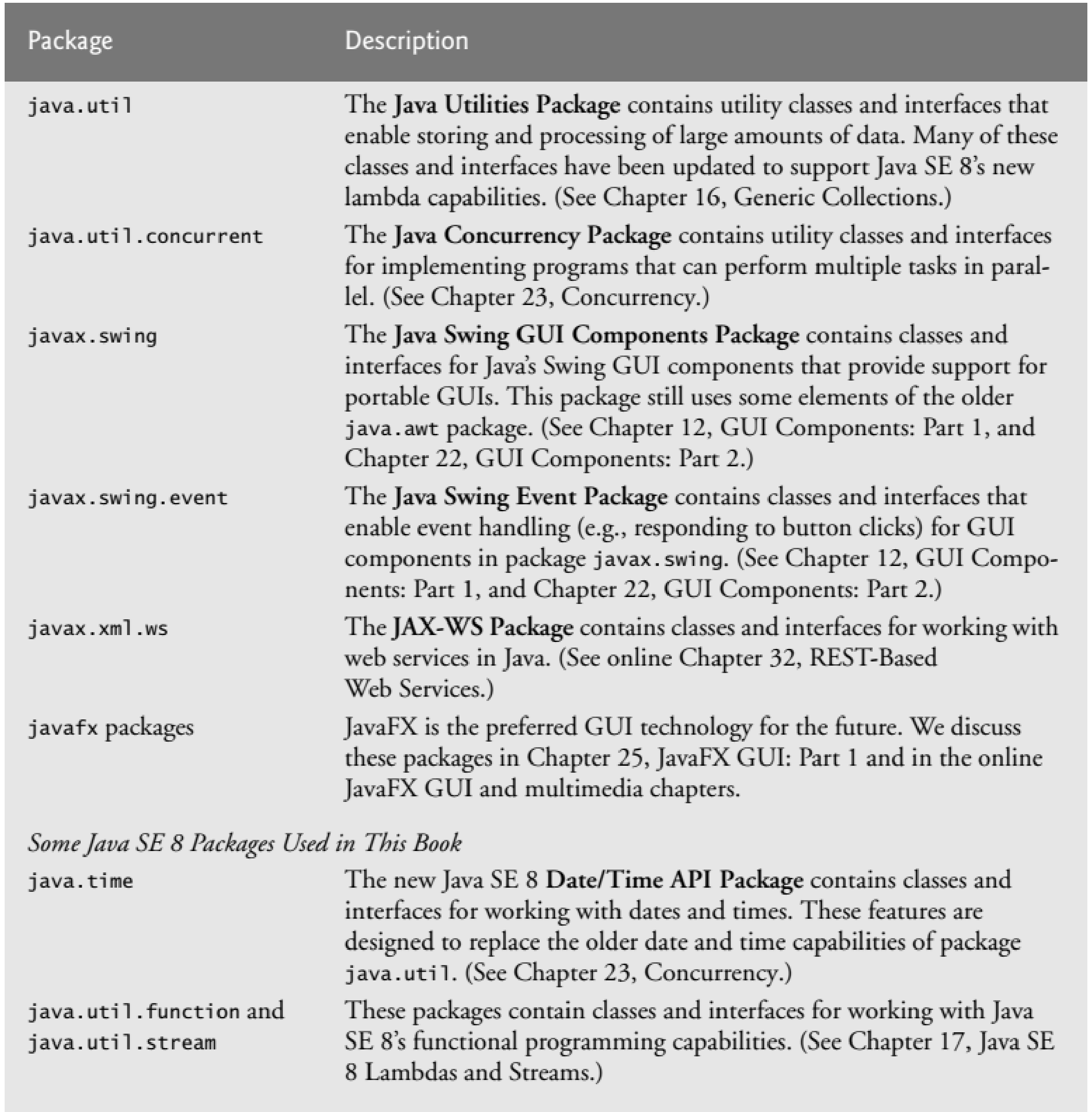
Java API Packages



Package	Description
`java.awt.event`	The **Java Abstract Window Toolkit Event Package** contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 13, Graphics and Java 2D.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (See Chapter 15, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed throughout the book) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See online Chapter 28, Networking.)
`java.security`	The **Java Security Package** contains classes and interfaces for enhancing application security.
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 24, Accessing Databases with JDBC.)

Fig. 5.5 | Java API packages (a subset). (Part 1 of 2.)

Java API Packages



Package	Description
`java.util`	The **Java Utilities Package** contains utility classes and interfaces that enable storing and processing of large amounts of data. Many of these classes and interfaces have been updated to support Java SE 8's new lambda capabilities. (See Chapter 16, Generic Collections.)
`java.util.concurrent`	The **Java Concurrency Package** contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 23, Concurrency.)
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. This package still uses some elements of the older `java.awt` package. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
`javax.swing.event`	The **Java Swing Event Package** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (See Chapter 12, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2.)
`javax.xml.ws`	The **JAX-WS Package** contains classes and interfaces for working with web services in Java. (See online Chapter 32, REST-Based Web Services.)
`javafx` packages	JavaFX is the preferred GUI technology for the future. We discuss these packages in Chapter 25, JavaFX GUI: Part 1 and in the online JavaFX GUI and multimedia chapters.
Some Java SE 8 Packages Used in This Book	
`java.time`	The new Java SE 8 **Date/Time API Package** contains classes and interfaces for working with dates and times. These features are designed to replace the older date and time capabilities of package `java.util`. (See Chapter 23, Concurrency.)
`java.util.function` and `java.util.stream`	These packages contain classes and interfaces for working with Java SE 8's functional programming capabilities. (See Chapter 17, Java SE 8 Lambdas and Streams.)

Fig. 5.5 | Java API packages (a subset). (Part 2 of 2.)

- Scope of Declaration

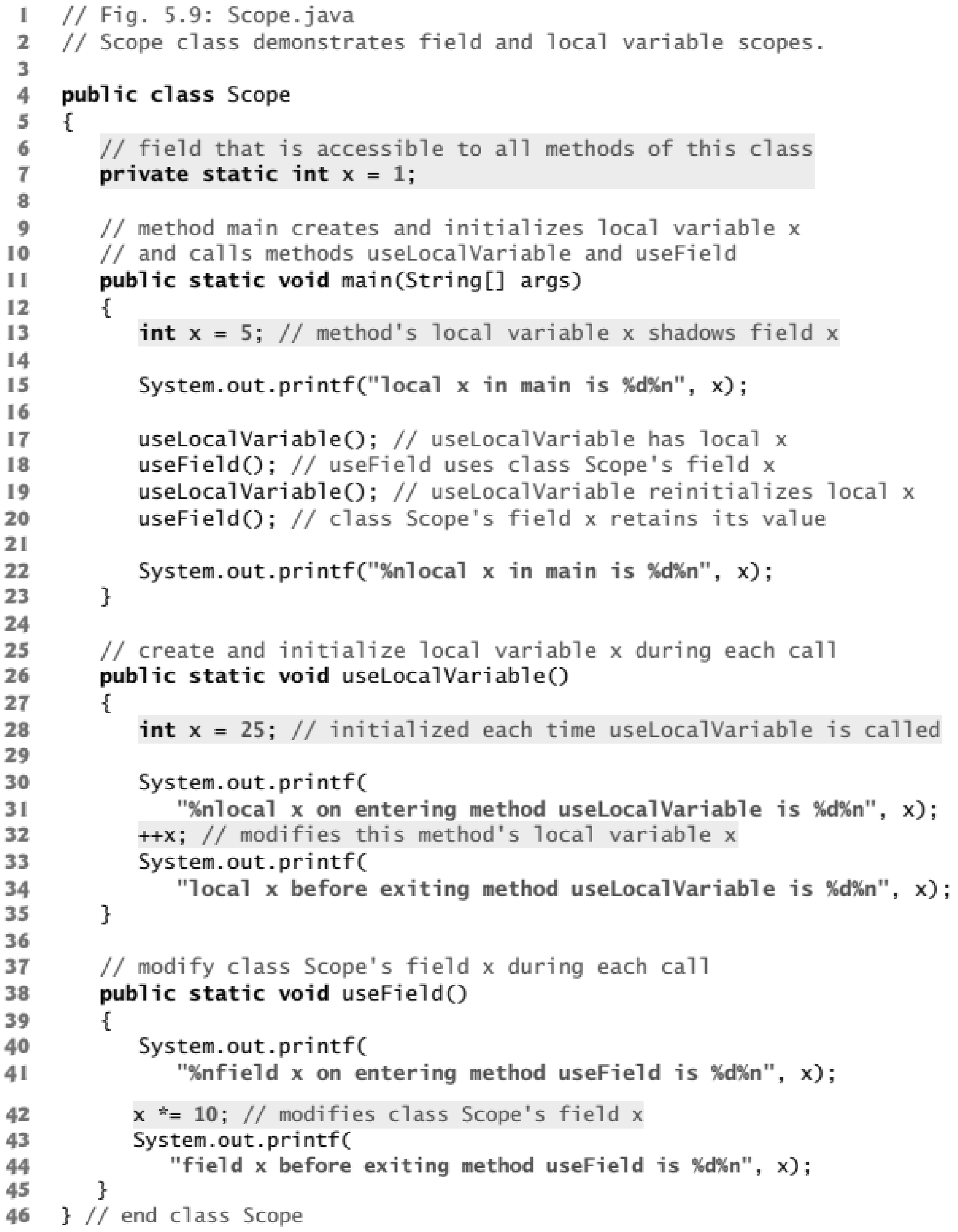
Scope of Declaration

- The scope of a declaration is the portion of the program that can refer to the declared entity by its name.
  - Portion of the program that can reference an entity by its name

  - Basic scope rules
    - Scope of a parameter declaration
    - Scope of a local-variable declaration
    - Scope of a label in a labeled `break` or `continue` statement
    - Scope of a local-variable declaration that appears in the initialization section of a `for` statement's header
    - Scope of a method or field of a class

Scope of Declaration

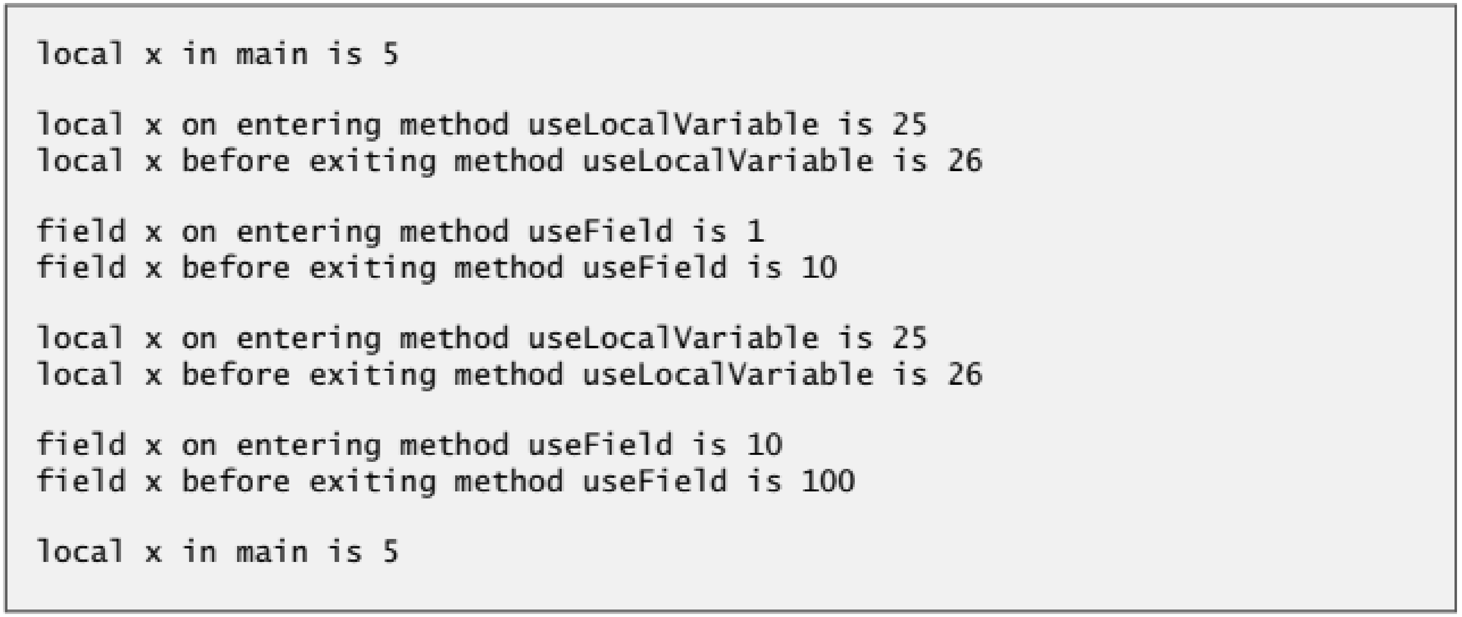
- Example:



```java
// Fig. 5.9: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private static int x = 1;

   // method main creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public static void main(String[] args)
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf("local x in main is %d%n", x);

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf("%nlocal x in main is %d%n", x);
   }

   // create and initialize local variable x during each call
   public static void useLocalVariable()
   {
      int x = 25; // initialized each time useLocalVariable is called

      System.out.printf(
         "%nlocal x on entering method useLocalVariable is %d%n", x);
      ++x; // modifies this method's local variable x
      System.out.printf(
         "local x before exiting method useLocalVariable is %d%n", x);
   }

   // modify class Scope's field x during each call
   public static void useField()
   {
      System.out.printf(
         "%nfield x on entering method useField is %d%n", x);
      x *= 10; // modifies class Scope's field x
      System.out.printf(
         "field x before exiting method useField is %d%n", x);
   }
} // end class Scope
```

Scope of Declaration

- Example:



```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

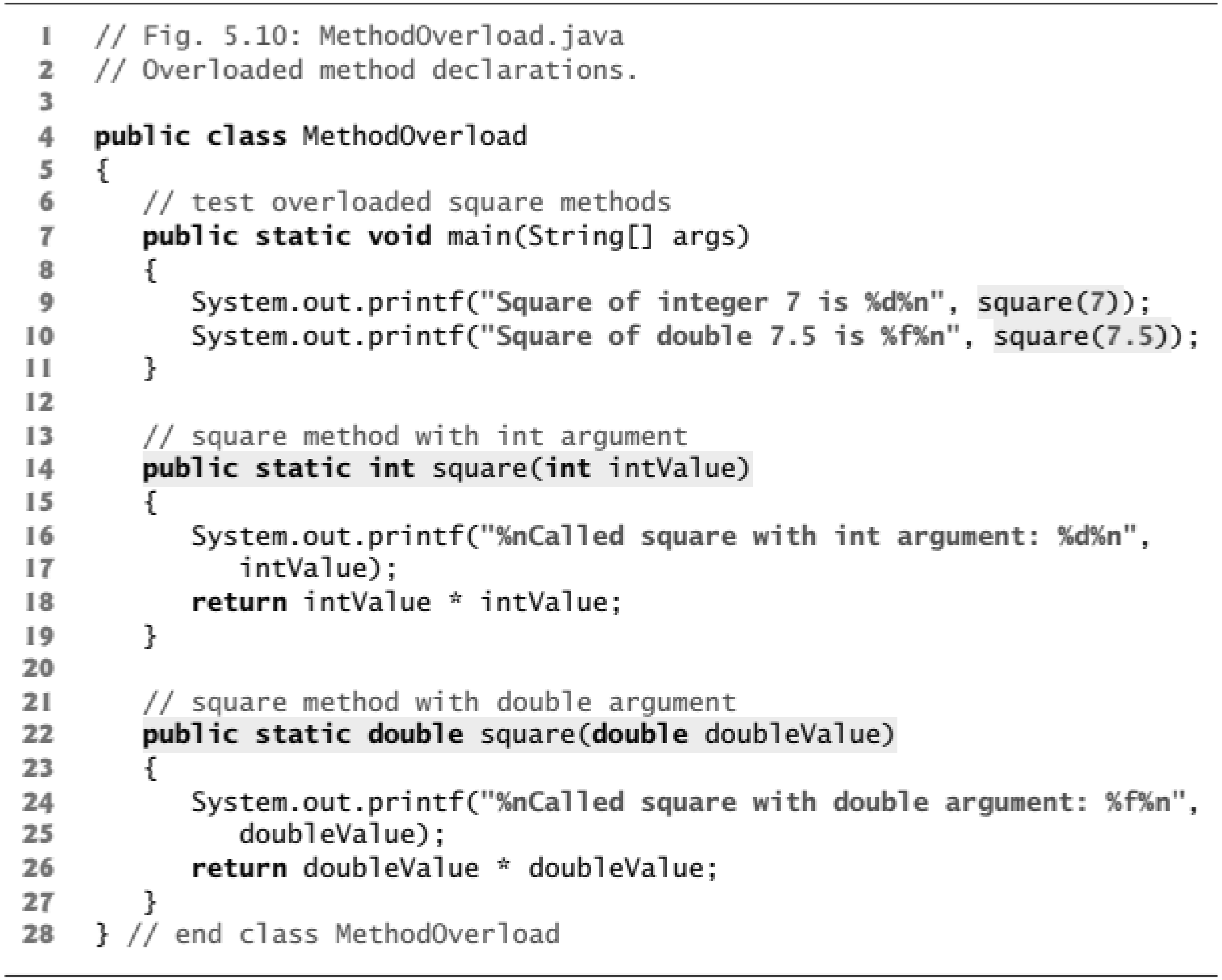
- Method Overloading

Method Overloading

- Several methods of the same name

- Different parameter set for each method
  - Number of parameters
  - Parameter types

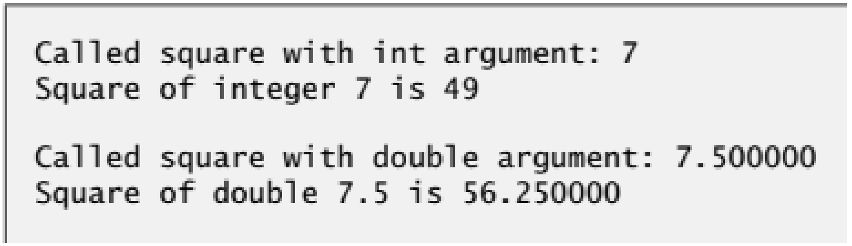
Method Overloading



```java
// Fig. 5.10: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public static void main(String[] args)
   {
      System.out.printf("Square of integer 7 is %d%n", square(7));
      System.out.printf("Square of double 7.5 is %f%n", square(7.5));
   }

   // square method with int argument
   public static int square(int intValue)
   {
      System.out.printf("%nCalled square with int argument: %d%n",
         intValue);
      return intValue * intValue;
   }

   // square method with double argument
   public static double square(double doubleValue)
   {
      System.out.printf("%nCalled square with double argument: %f%n",
         doubleValue);
      return doubleValue * doubleValue;
   }
} // end class MethodOverload
```

Fig. 5.10 | Overloaded method declarations. (Part 1 of 2.)

Output



```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Method Overloading

- Example:

```
class Adder{
        static int add(int a, int b){
                return a+b;
        }
        static double add(double a, double b){
                return a+b;
        }
        public static void main(String[] args){
                System.out.println(add(11,11));
                System.out.println(add(12.3,12.6));
        }
}
```

Reference: https://www.javatpoint.com/method-overloading-in-java

Method Overloading

- Example:

```
class Adder{
	static int add(int a,int b){
		return a+b;
	}
	static int add(int a,int b,int c){
		return a+b+c;
	}

	public static void main(String[] args){
		System.out.println(add(11,11));
		System.out.println(add(11,11,11));
	}
}
```

Reference: https://www.javatpoint.com/method-overloading-in-java

Method Overloading

- Example of Method Overloading with Type Promotion

```
class OverloadingCalculation1{
        static void sum(int a,long b){
                System.out.println(a+b);
        }
        static void sum(int a,int b,int c){
                System.out.println(a+b+c);
        }

 public static void main(String args[]){
         sum(20,20);//now second int literal will be promoted to long
         sum(20,20,20);
         }
}
```

Reference: https://www.javatpoint.com/method-overloading-in-java

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming
Week 9
Arrays

Contents

1. Primitive vs. Reference type
2. Creating, Initializing, and modifying arrays
3. Exception Handling (Try-Catch Statement)
4. Enhanced for Statement
5. Passing Arrays to Methods
6. Pass-By-Value vs. Pass-By-Reference
7. Multidimensional Arrays

Weekly Learning Outcomes

- Make use of Arrays to solve different Java problems
- Differentiate between Pass-By-Value vs. Pass-By-Reference

Required Reading

1. Chapter 6 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 3 - (Java The Complete Reference, Eleventh Edition)
2. Java Arrays Tutorial:

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

- Primitive vs. Reference type

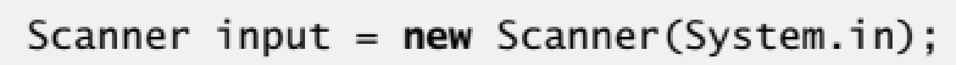
Primitive vs. Reference type

- A primitive-type variable
  - can hold exactly *one* value of its declared type at a time. For example, an int variable can store one integer at a time.

  - When another value is assigned to that variable, the new value replaces the previous one.

Primitive vs. Reference type

- Programs use variables of reference types
    - to store the *addresses* of objects in the computer's memory.

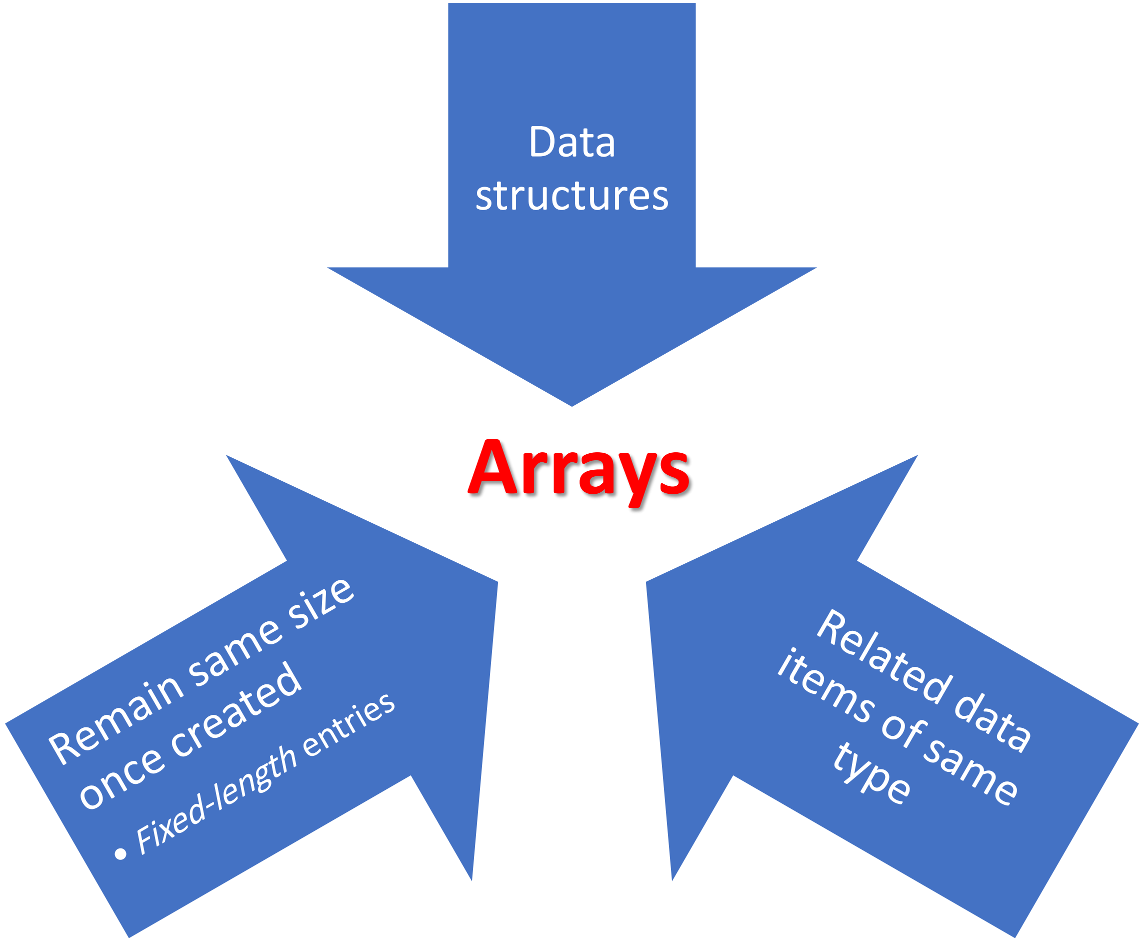


```
Scanner input = new Scanner(System.in);
```

- creates a Scanner object, then assigns to the variable input a *reference* to that Scanner object.

- Creating, Initializing, and modifying arrays

Creating, Initializing, and modifying arrays



Data structures

Arrays

Remain same size once created

- *Fixed-length* entries

Related data items of same type

Creating, Initializing, and modifying arrays

- Arrays

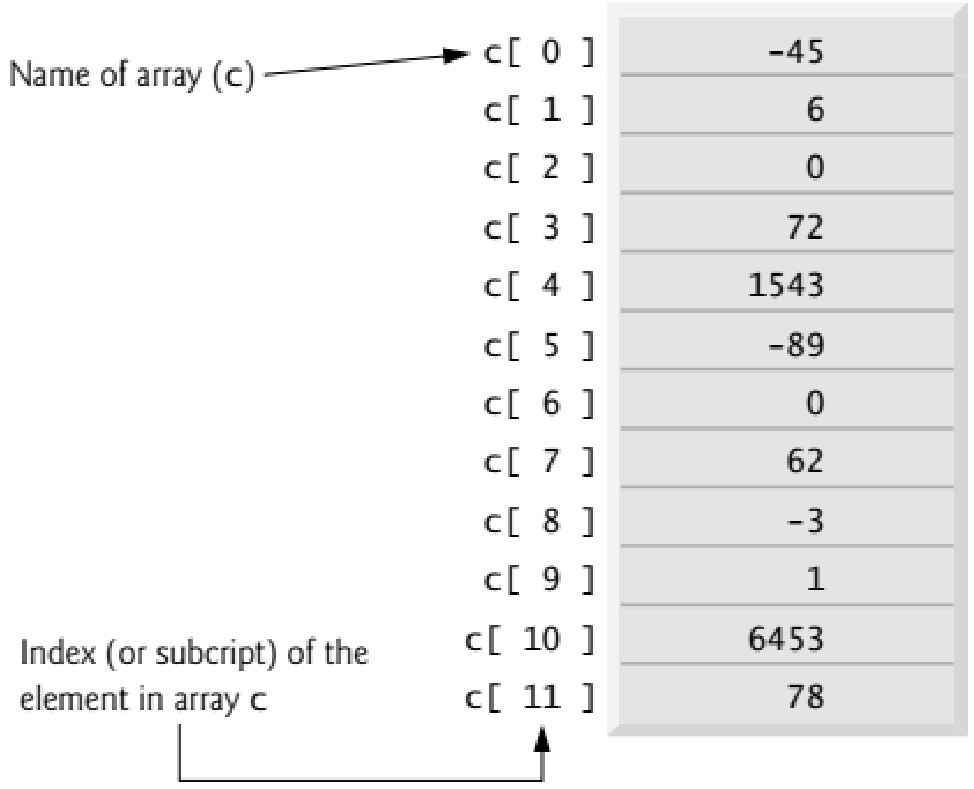


Fig. 6.1 | A 12-element array.

Creating, Initializing, and modifying arrays

- Index
  - Also called subscript
  - Position number in square brackets
  - Must be positive integer or integer expression

```
a = 5;
b = 6;
c[ a + b ] += 2;
```

- Adds `2` to `c[ 11 ]`

Creating, Initializing, and modifying arrays

- The name of the array is c
- c.length returns array c’s length
- This array’s 12 elements are referred to as c[0], c[1], c[2], …, c[11]
- The value of c[0] is -45
- The value of c[1] is 6
- The value of c[2] is 0

- sum = c[**0**] + c[**1**] + c[**2**];

Creating, Initializing, and modifying arrays

- **Declaring and Creating Arrays**
  - Array objects occupy space in memory.
  - Arrays are created with keyword new.

```
int[] c = new int[12];
```

  - Equivalent to:

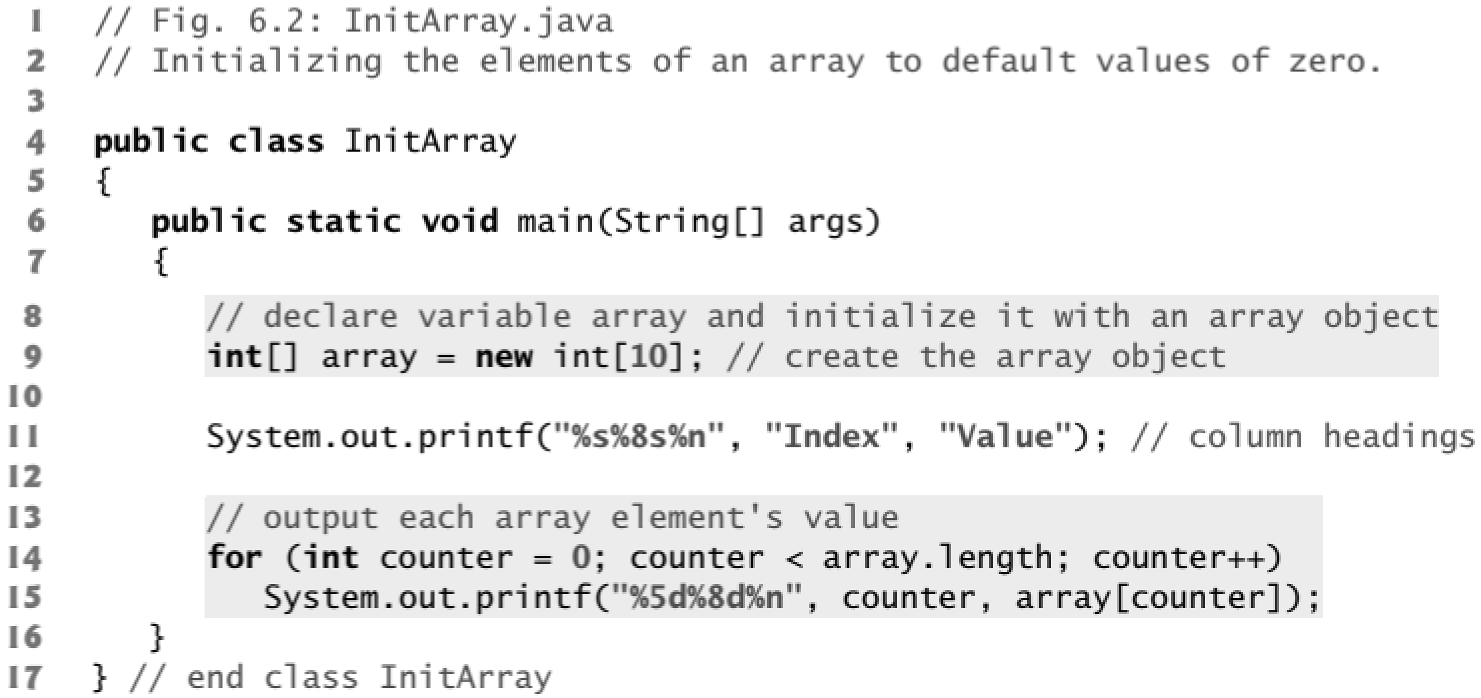
```
int[] c;                // declare the array variable
c = new int[12]; // create the array; assign to array variable
```

  - Create several arrays in a single declaration

```
String[] b = new String[100], x = new String[27];
```

Creating, Initializing, and modifying arrays

- **Creating and Initializing an Array Example**



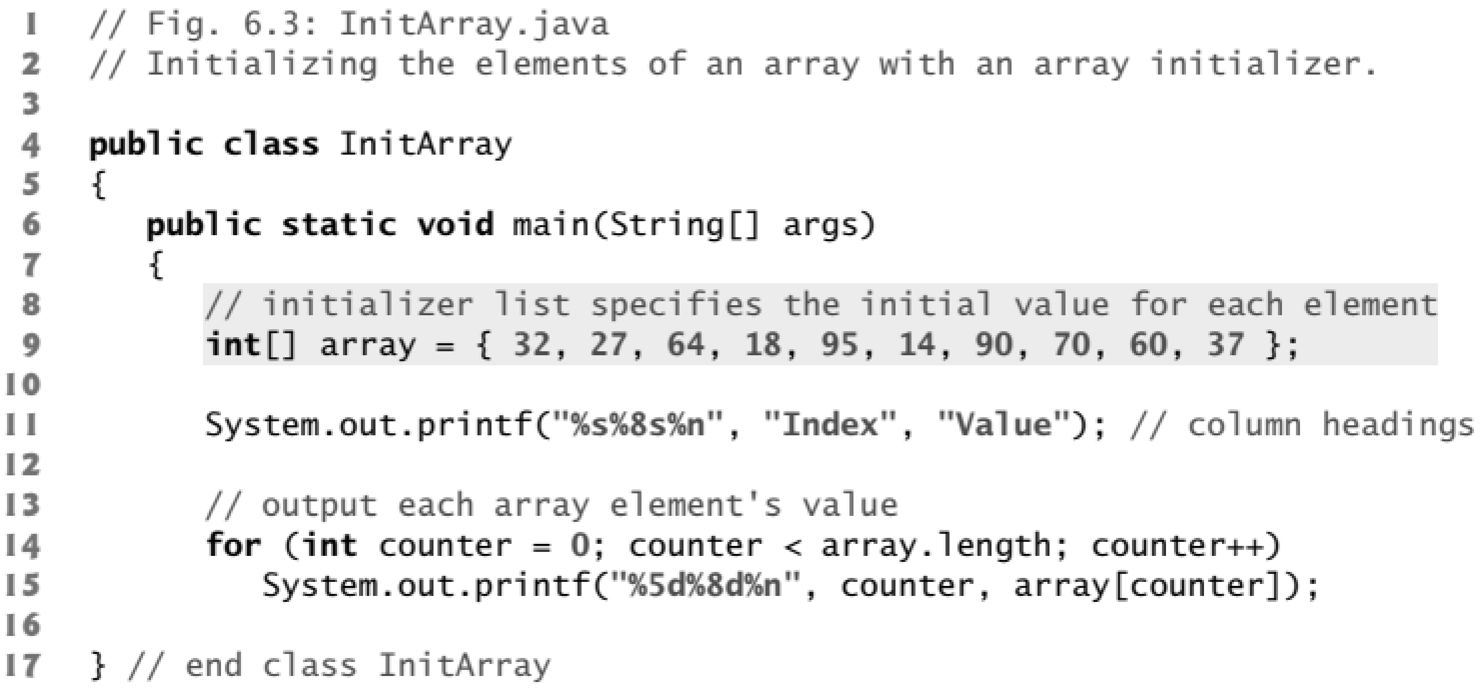
```
// Fig. 6.2: InitArray.java
// Initializing the elements of an array to default values of zero.

public class InitArray
{
   public static void main(String[] args)
   {
      // declare variable array and initialize it with an array object
      int[] array = new int[10]; // create the array object

      System.out.printf("%s%8s%n", "Index", "Value"); // column headings

      // output each array element's value
      for (int counter = 0; counter < array.length; counter++)
         System.out.printf("%5d%8d%n", counter, array[counter]);
   }
} // end class InitArray
```

Output

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Creating, Initializing, and modifying arrays

- **Using an Array Initializer**



```java
// Fig. 6.3: InitArray.java
// Initializing the elements of an array with an array initializer.

public class InitArray
{
   public static void main(String[] args)
   {
      // initializer list specifies the initial value for each element
      int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

      System.out.printf("%s%8s%n", "Index", "Value"); // column headings

      // output each array element's value
      for (int counter = 0; counter < array.length; counter++)
         System.out.printf("%5d%8d%n", counter, array[counter]);

} // end class InitArray
```

Output

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Creating, Initializing, and modifying arrays

- **Calculating the Values to Store in an Array**

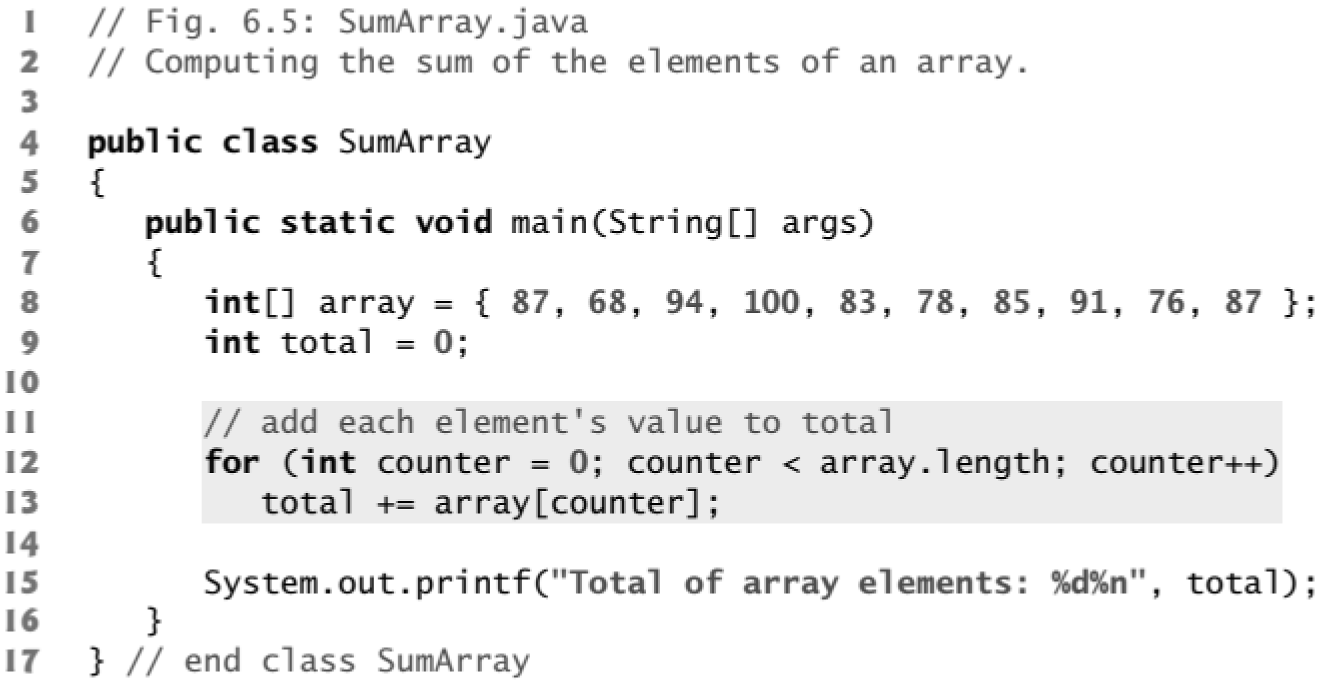
```java
// Fig. 6.4: InitArray.java
// Calculating the values to be placed into the elements of an array.

public class InitArray
{
   public static void main(String[] args)
   {
      final int ARRAY_LENGTH = 10; // declare constant
      int[] array = new int[ARRAY_LENGTH]; // create array

      // calculate value for each array element
      for (int counter = 0; counter < array.length; counter++)
         array[counter] = 2 + 2 * counter;

      System.out.printf("%s%8s%n", "Index", "Value"); // column headings

      // output each array element's value
      for (int counter = 0; counter < array.length; counter++)
         System.out.printf("%5d%8d%n", counter, array[counter]);
   }
} // end class InitArray
```

Output

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Creating, Initializing, and modifying arrays

- **Summing the Elements of an Array**



```java
// Fig. 6.5: SumArray.java
// Computing the sum of the elements of an array.

public class SumArray
{
   public static void main(String[] args)
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for (int counter = 0; counter < array.length; counter++)
         total += array[counter];

      System.out.printf("Total of array elements: %d%n", total);
   }
} // end class SumArray
```

Output

```
Total of array elements: 849
```

Creating, Initializing, and modifying arrays

- **Using Bar Charts to Display Array Data Graphically**

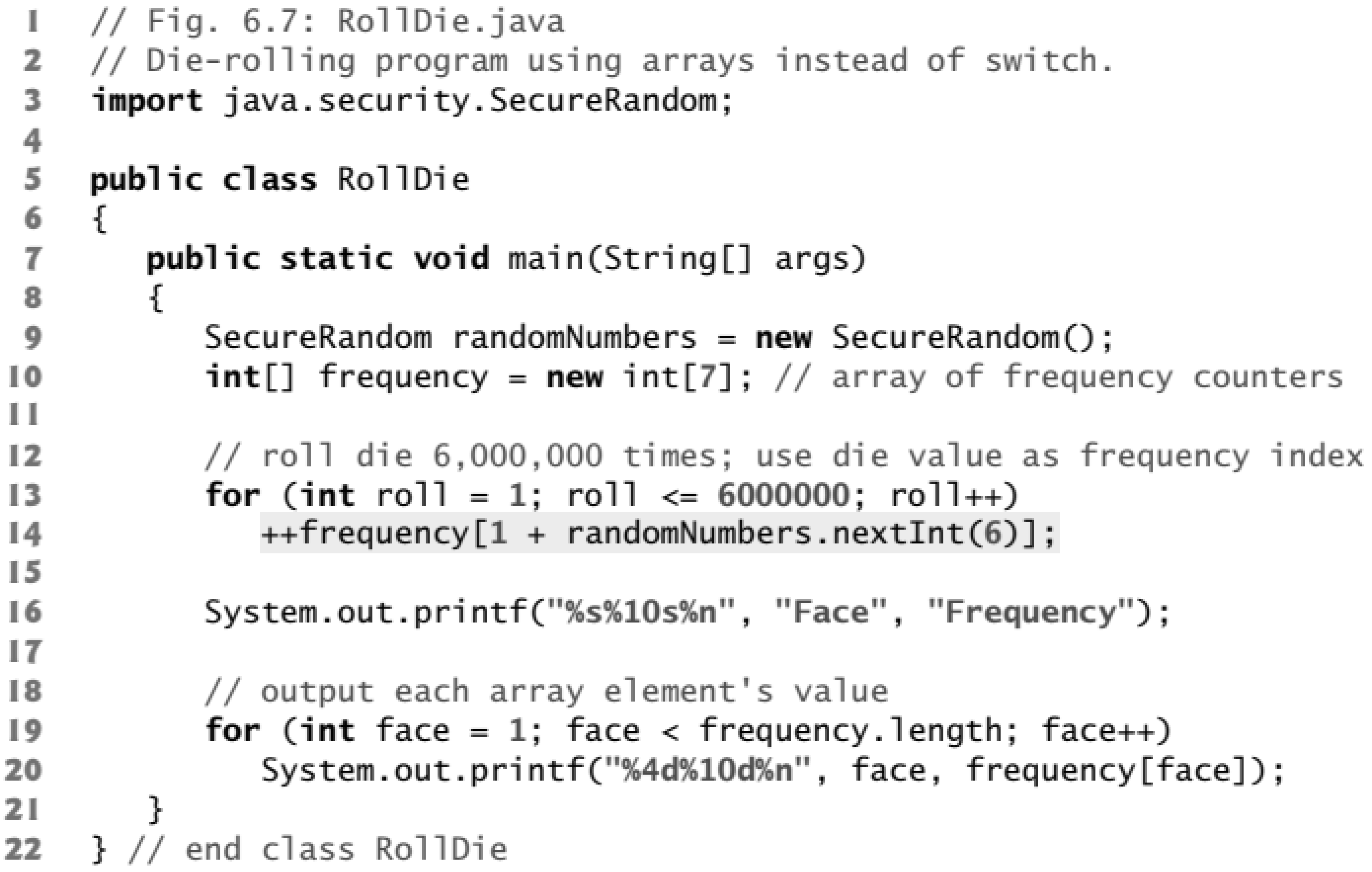
```java
// Fig. 6.6: BarChart.java
// Bar chart printing program.

public class BarChart
{
   public static void main(String[] args)
   {
      int[] array = {0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};

      System.out.println("Grade distribution:");

      // for each array element, output a bar of the chart
      for (int counter = 0; counter < array.length; counter++)
      {
         // output bar label ("00-09: ", ..., "90-99: ", "100: ")
         if (counter == 10)
            System.out.printf("%5d: ", 100);
         else
            System.out.printf("%02d-%02d: ",
               counter * 10, counter * 10 + 9);

         // print bar of asterisks
         for (int stars = 0; stars < array[counter]; stars++)
            System.out.print("*");

         System.out.println();
      }
   }
} // end class BarChart
```

Output

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

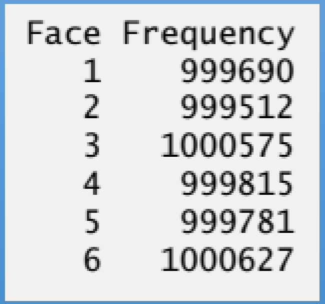
Creating, Initializing, and modifying arrays

- **Using the Elements of an Array as Counters**



```java
// Fig. 6.7: RollDie.java
// Die-rolling program using arrays instead of switch.
import java.security.SecureRandom;

public class RollDie
{
   public static void main(String[] args)
   {
      SecureRandom randomNumbers = new SecureRandom();
      int[] frequency = new int[7]; // array of frequency counters

      // roll die 6,000,000 times; use die value as frequency index
      for (int roll = 1; roll <= 6000000; roll++)
         ++frequency[1 + randomNumbers.nextInt(6)];

      System.out.printf("%s%10s%n", "Face", "Frequency");

      // output each array element's value
      for (int face = 1; face < frequency.length; face++)
         System.out.printf("%4d%10d%n", face, frequency[face]);
   }
} // end class RollDie
```

Output



```
Face Frequency
   1    999690
   2    999512
   3   1000575
   4    999815
   5    999781
   6   1000627
```

- Exception Handling (Try-Catch Statement)

Exception Handling (Try-Catch Statement)

An exception indicates

- A problem that occurs while a program executes.
- To handle an exception, place any code that might throw an exception in a **try-Catch** statement.

The try Statement

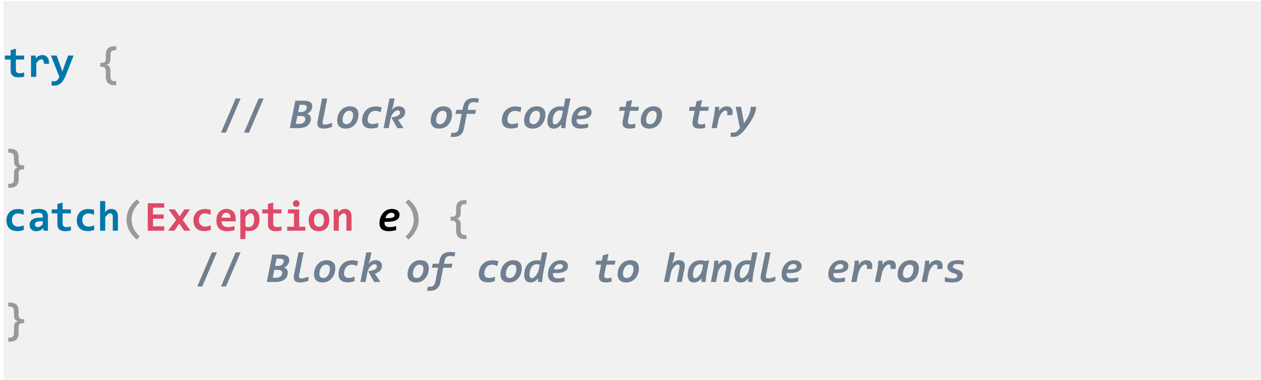
- contains the code that might *throw* an exception.

The Catch Statement

- contains the code that *handles* the exception if one occurs.

Exception Handling (Try-Catch Statement)

- Try-Catch Syntax



```
try {
        // Block of code to try
}
catch(Exception e) {
       // Block of code to handle errors
}
```

Reference: https://www.w3schools.com/java/java_try_catch.asp

Exception Handling (Try-Catch Statement)

- Exception Example

```
public class Main {
        public static void main(String[ ] args) {
                int[] myNumbers = {1, 2, 3};
                System.out.println(myNumbers[10]); // error!
        }
}
```

Output

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Main.main(Main.java:4)
```

Reference: https://www.w3schools.com/java/java_try_catch.asp

Exception Handling (Try-Catch Statement)

- finally statement lets you execute code, after try...catch, regardless of the result:

```
public class Main {
        public static void main(String[ ] args) {
                try {
                        int[] myNumbers = {1, 2, 3};
                        System.out.println(myNumbers[10]);
                }
                catch (Exception e) {
                        System.out.println("Something went wrong.");
                }
                finally {
                        System.out.println(“The try catch is finished.");
                }
        }
}
```

Output

```
Something went wrong.
```

Reference: https://www.w3schools.com/java/java_try_catch.asp

Exception Handling (Try-Catch Statement)

- Exception Example

```
public class Main {
        public static void main(String[ ] args) {
                try {
                        int[] myNumbers = {1, 2, 3};
                        System.out.println(myNumbers[10]);
                }
                catch (Exception e) {
                        System.out.println("Something went wrong.");
                }
        }
}
```

Output

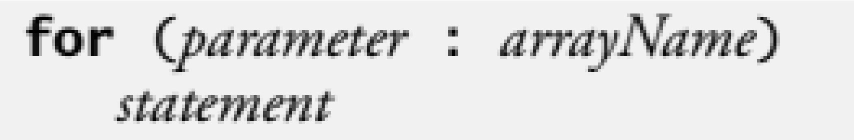
```
Something went wrong.
```

Reference: https://www.w3schools.com/java/java_try_catch.asp

- Enhanced for Statement

Enhanced for Statement

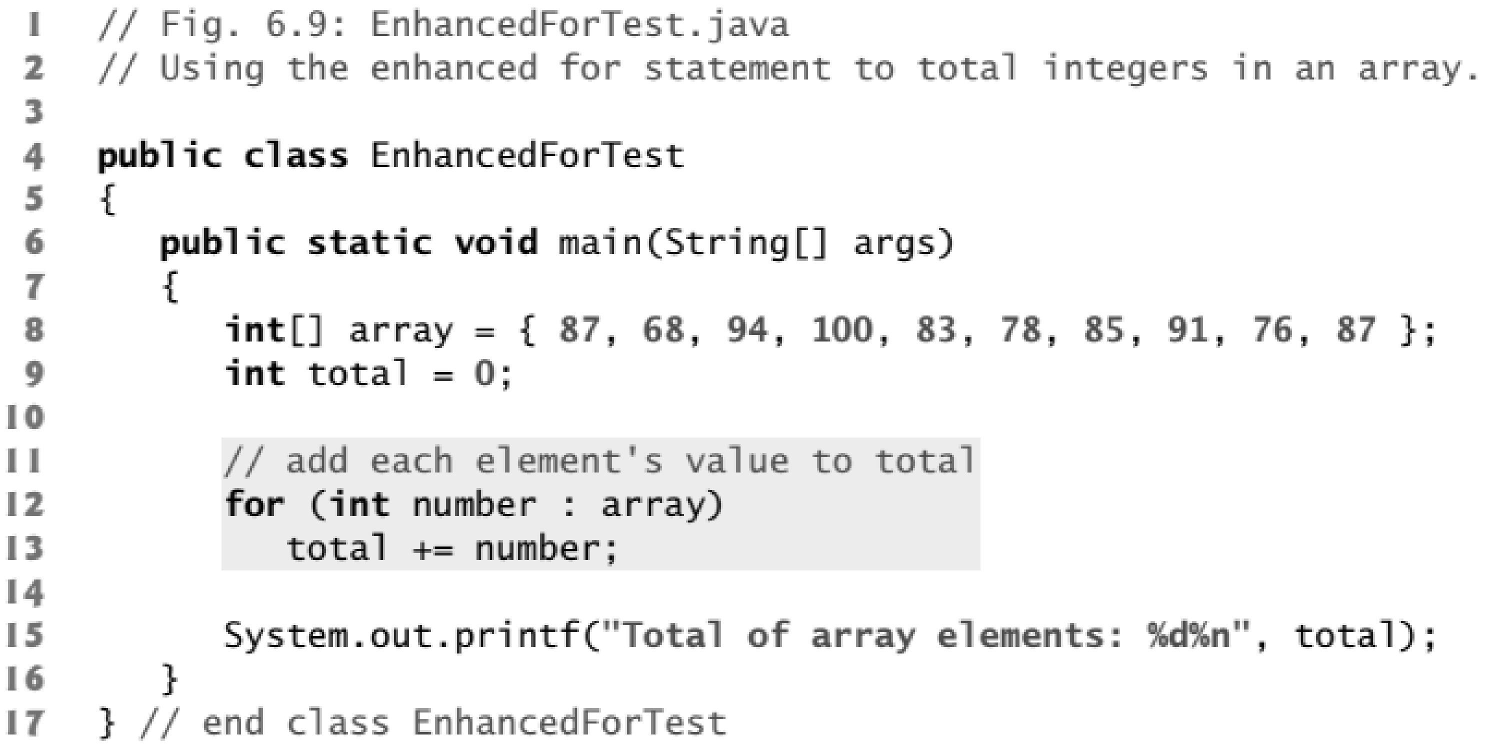
- Iterates through the elements of an array *without* using a counter.
  - thus avoiding the possibility of “stepping outside” the array



```
for (parameter : arrayName)
    statement
```

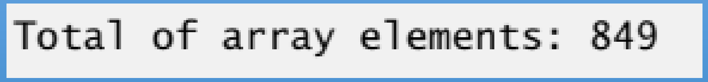
  - *parameter* has a *type* and an *identifier*
  - *arrayName* is the array through which to iterate.

Enhanced for Statement



```
// Fig. 6.9: EnhancedForTest.java
// Using the enhanced for statement to total integers in an array.

public class EnhancedForTest
{
   public static void main(String[] args)
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      for (int number : array)
         total += number;

      System.out.printf("Total of array elements: %d%n", total);
   }
} // end class EnhancedForTest
```

Output

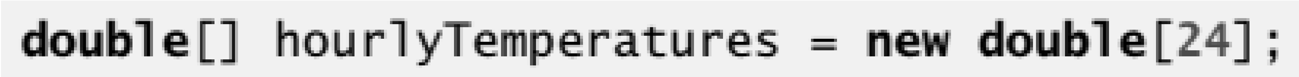


```
Total of array elements: 849
```

- Passing Arrays to Methods

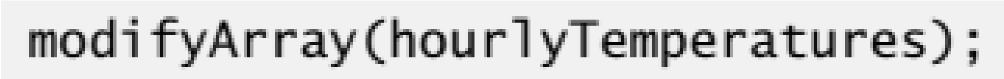
Passing Arrays to Methods

- To pass an array argument to a method, specify the name of the array *without any brackets*.
  - For example to pass the array hourlyTemperatures



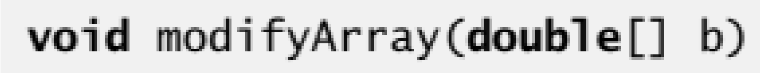
```
double[] hourlyTemperatures = new double[24];
```

  - The method call will be:



```
modifyArray(hourlyTemperatures);
```

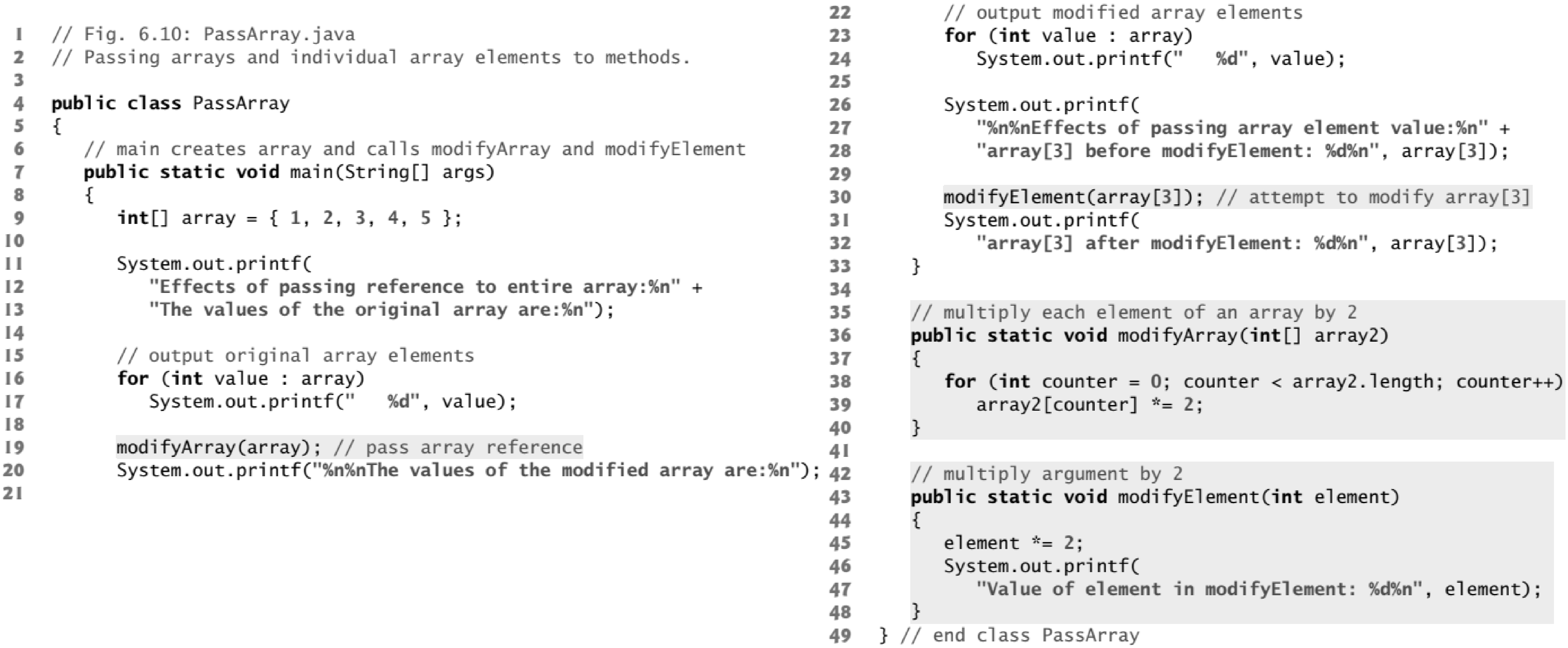
  - The method header will be:



```
void modifyArray(double[] b)
```

Passing Arrays to Methods

- Example:

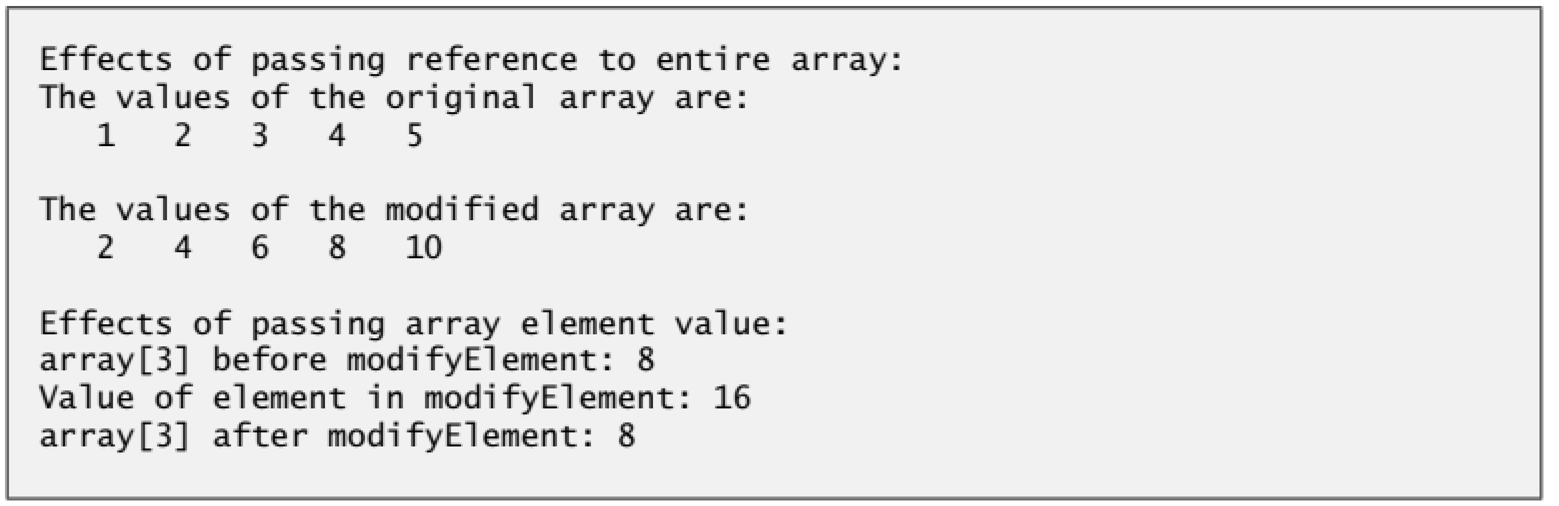


```java
// Fig. 6.10: PassArray.java
// Passing arrays and individual array elements to methods.

public class PassArray
{
   // main creates array and calls modifyArray and modifyElement
   public static void main(String[] args)
   {
      int[] array = { 1, 2, 3, 4, 5 };

      System.out.printf(
         "Effects of passing reference to entire array:%n" +
         "The values of the original array are:%n");

      // output original array elements
      for (int value : array)
         System.out.printf("   %d", value);

      modifyArray(array); // pass array reference
      System.out.printf("%n%nThe values of the modified array are:%n");

      // output modified array elements
      for (int value : array)
         System.out.printf("   %d", value);

      System.out.printf(
         "%n%nEffects of passing array element value:%n" +
         "array[3] before modifyElement: %d%n", array[3]);

      modifyElement(array[3]); // attempt to modify array[3]
      System.out.printf(
         "array[3] after modifyElement: %d%n", array[3]);
   }

   // multiply each element of an array by 2
   public static void modifyArray(int[] array2)
   {
      for (int counter = 0; counter < array2.length; counter++)
         array2[counter] *= 2;
   }

   // multiply argument by 2
   public static void modifyElement(int element)
   {
      element *= 2;
      System.out.printf(
         "Value of element in modifyElement: %d%n", element);
   }
} // end class PassArray
```

Passing Arrays to Methods

- Example:

Output



```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

- Pass-By-Value vs. Pass-By-Reference

Pass-By-Value vs. Pass-By-Reference

- Two ways to pass arguments in method calls in many programming languages are
  - pass-by-value (call-by-value).
  - pass-by-reference (call-by-reference).

Pass-By-Value vs. Pass-By-Reference

- pass-by-value
  - When an argument is passed by value:
    - a *copy* of the argument’s *value* is passed to the called method.
    - The called method works exclusively with the copy.
    - Changes to the called method’s copy do *not* affect the original variable’s value in the caller

Pass-By-Value vs. Pass-By-Reference

- pass-by-reference
  - When an argument is passed by reference:
    - The called method can access the argument’s value in the caller directly and modify that data, if necessary.
    - It improves performance by eliminating the need to copy possibly large amounts of data.

Pass-By-Value vs. Pass-By-Reference

Java does *not* allow you to choose pass-by-value or pass-by-reference—*all arguments are passed by value*.

- Dealing with reference object type will be discussed in detail in later chapters.

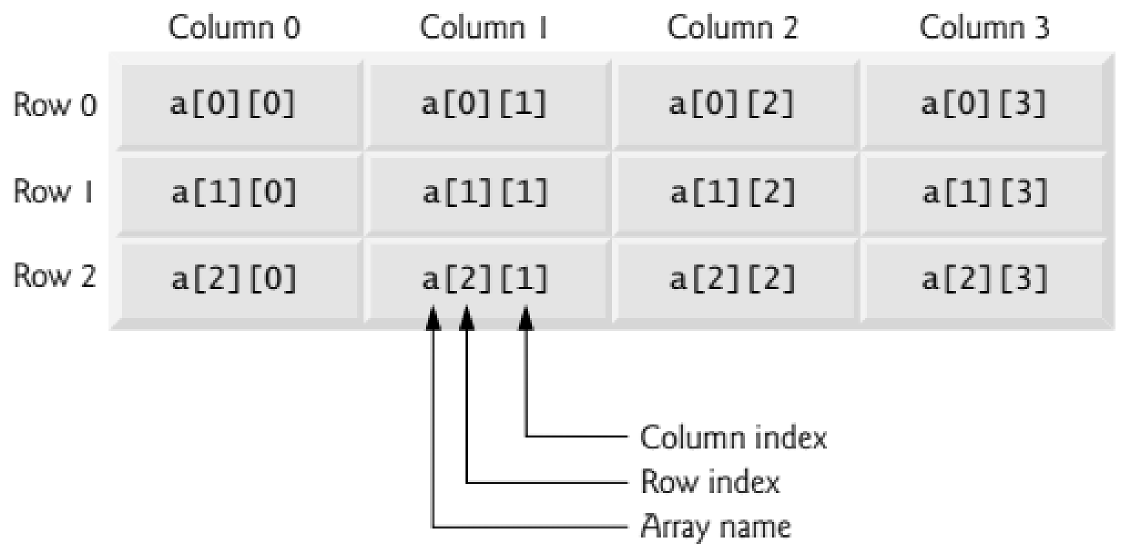
- Multidimensional Arrays

Multidimensional Arrays

Multidimensional arrays

- Tables with rows and columns
  - Two-dimensional array
  - Declaring two-dimensional array b[2][2]
    - int b[][] = { { 1, 2 }, { 3, 4 } };
    - 1 and 2 initialize b[0][0] and b[0][1]
    - 3 and 4 initialize b[1][0] and b[1][1]
    - int b[][] = { { 1, 2 }, { 3, 4, 5 } };
    - row 0 contains elements 1 and 2
    - row 1 contains elements 3, 4 and 5

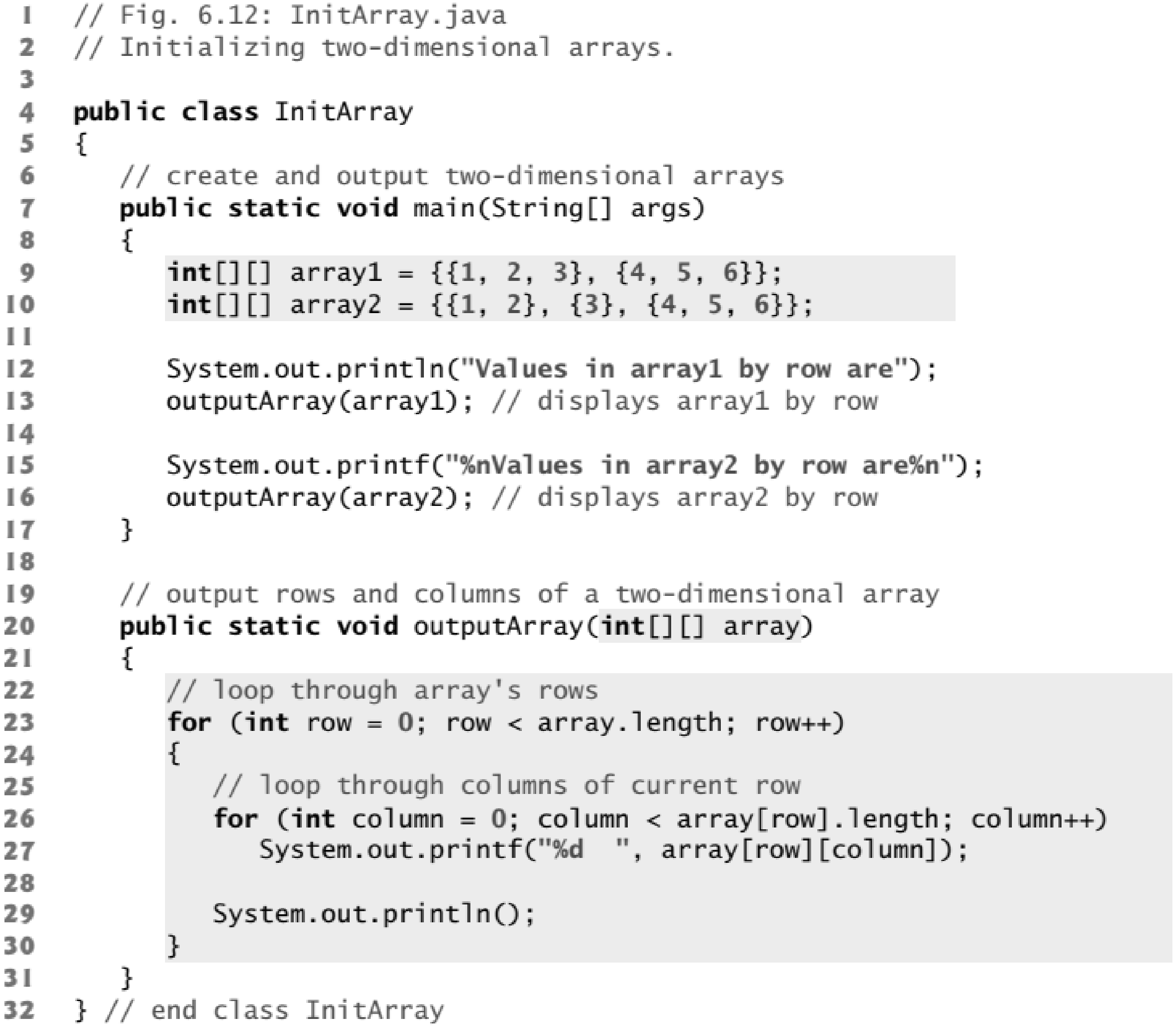
Multidimensional Arrays



	Column 0	Column 1	Column 2	Column 3
Row 0	`a[0][0]`	`a[0][1]`	`a[0][2]`	`a[0][3]`
Row 1	`a[1][0]`	`a[1][1]`	`a[1][2]`	`a[1][3]`
Row 2	`a[2][0]`	`a[2][1]`	`a[2][2]`	`a[2][3]`

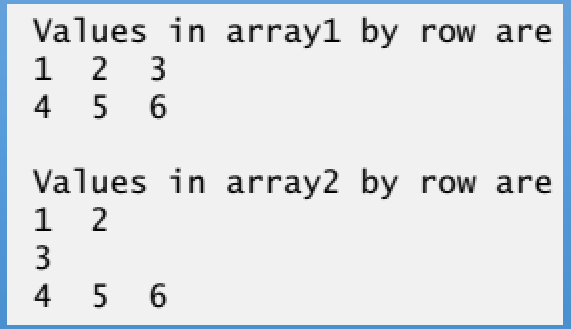
Fig. 6.11 | Two-dimensional array with three rows and four columns.

Multidimensional Arrays



```
// Fig. 6.12: InitArray.java
// Initializing two-dimensional arrays.

public class InitArray
{
   // create and output two-dimensional arrays
   public static void main(String[] args)
   {
      int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
      int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};

      System.out.println("Values in array1 by row are");
      outputArray(array1); // displays array1 by row

      System.out.printf("%nValues in array2 by row are%n");
      outputArray(array2); // displays array2 by row
   }

   // output rows and columns of a two-dimensional array
   public static void outputArray(int[][] array)
   {
      // loop through array's rows
      for (int row = 0; row < array.length; row++)
      {
         // loop through columns of current row
         for (int column = 0; column < array[row].length; column++)
            System.out.printf("%d  ", array[row][column]);

         System.out.println();
      }
   }
} // end class InitArray
```

Output



```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 10

Classes and Objects I

Contents

1. Declaring classes, instance variables, methods
2. Default and Explicit Initialization for Instance Variables
3. Constructors, private, public, static.
4. Instantiating and using objects
5. Implementing a test class

Weekly Learning Outcomes

- Construct Java Classes and Instantiating objects
- Plan and design a multi class application

Required Reading

1. Chapter 7 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 6 - (Java The Complete Reference, Eleventh Edition)
2. Java Classes and Objects:

https://www.w3schools.com/java/java_classes.asp

- Declaring classes, instance variables, methods

Declaring classes, instance variables, methods

Object Oriented Programming (OOP)

Encapsulates data (attributes) and methods (behaviors)

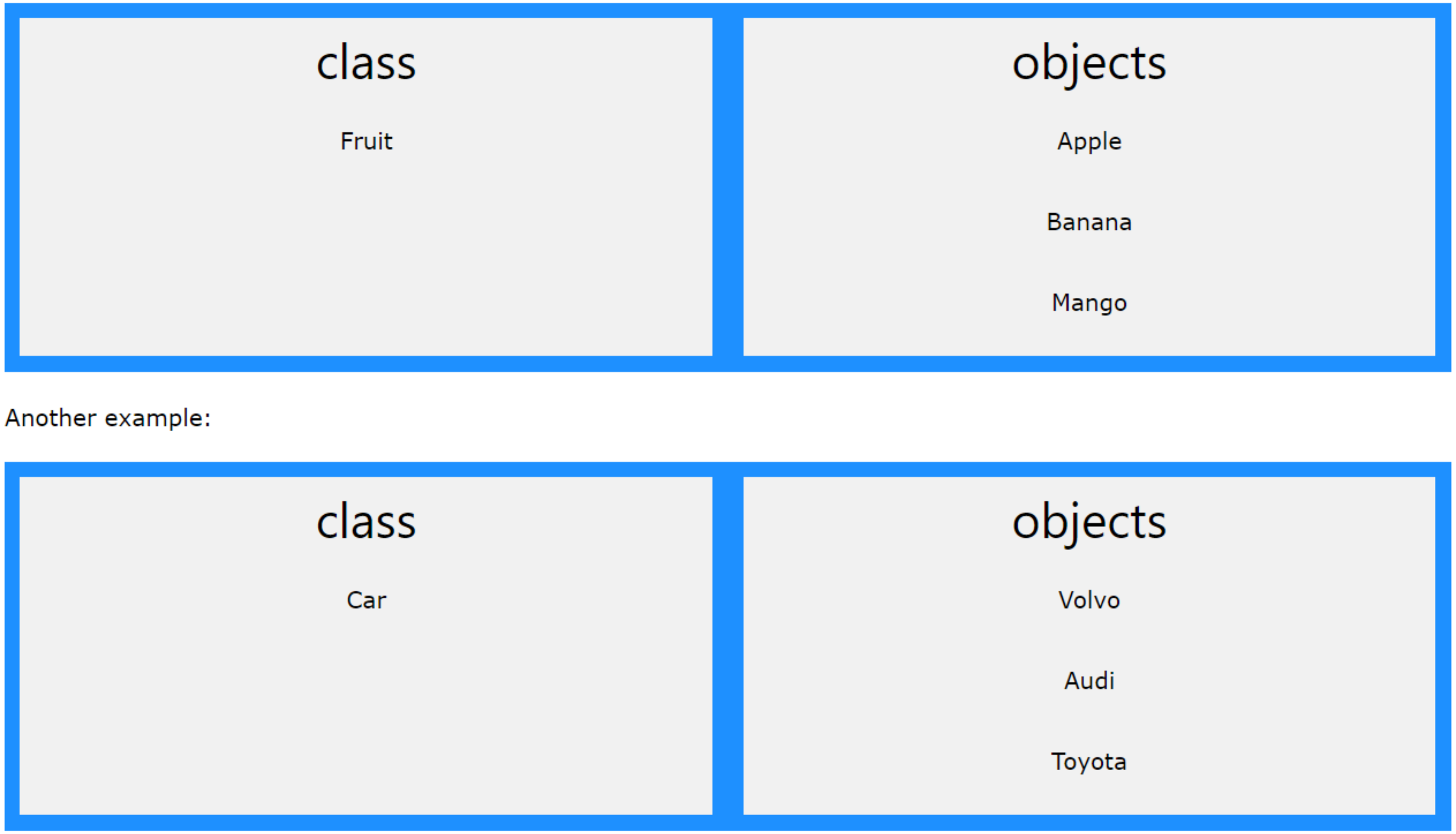
- Objects

Allows objects to communicate

- Well-defined *interfaces*

Declaring classes, instance variables, methods

- What are Classes and Objects?



class	objects
Fruit	Apple Banana Mango

Another example:

class	objects
Car	Volvo Audi Toyota

Reference: https://www.w3schools.com/java/java_oop.asp

Declaring classes, instance variables, methods

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

- A class in Java can contain:
  - **Fields**
  - **Methods**
  - **Constructors**
  - **Blocks**
  - **Nested class and interface**

Reference: https://www.javatpoint.com/object-and-class-in-java

Declaring classes, instance variables, methods

- **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

- **Object Definitions:**
  - An object is *a real-world entity*.
  - An object is *a runtime entity*.
  - The object is *an entity which has state and behavior*.
  - The object is *an instance of a class*.

Reference: https://www.javatpoint.com/object-and-class-in-java

Declaring classes, instance variables, methods

- Instance variable in Java
  - A variable which is created inside the class but outside the method is known as an instance variable.
  - Instance variable doesn't get memory at compile time.
  - It gets memory at runtime when an object or instance is created.
  - That is why it is known as an instance variable.

Reference: https://www.javatpoint.com/object-and-class-in-java

Declaring classes, instance variables, methods

Method in Java

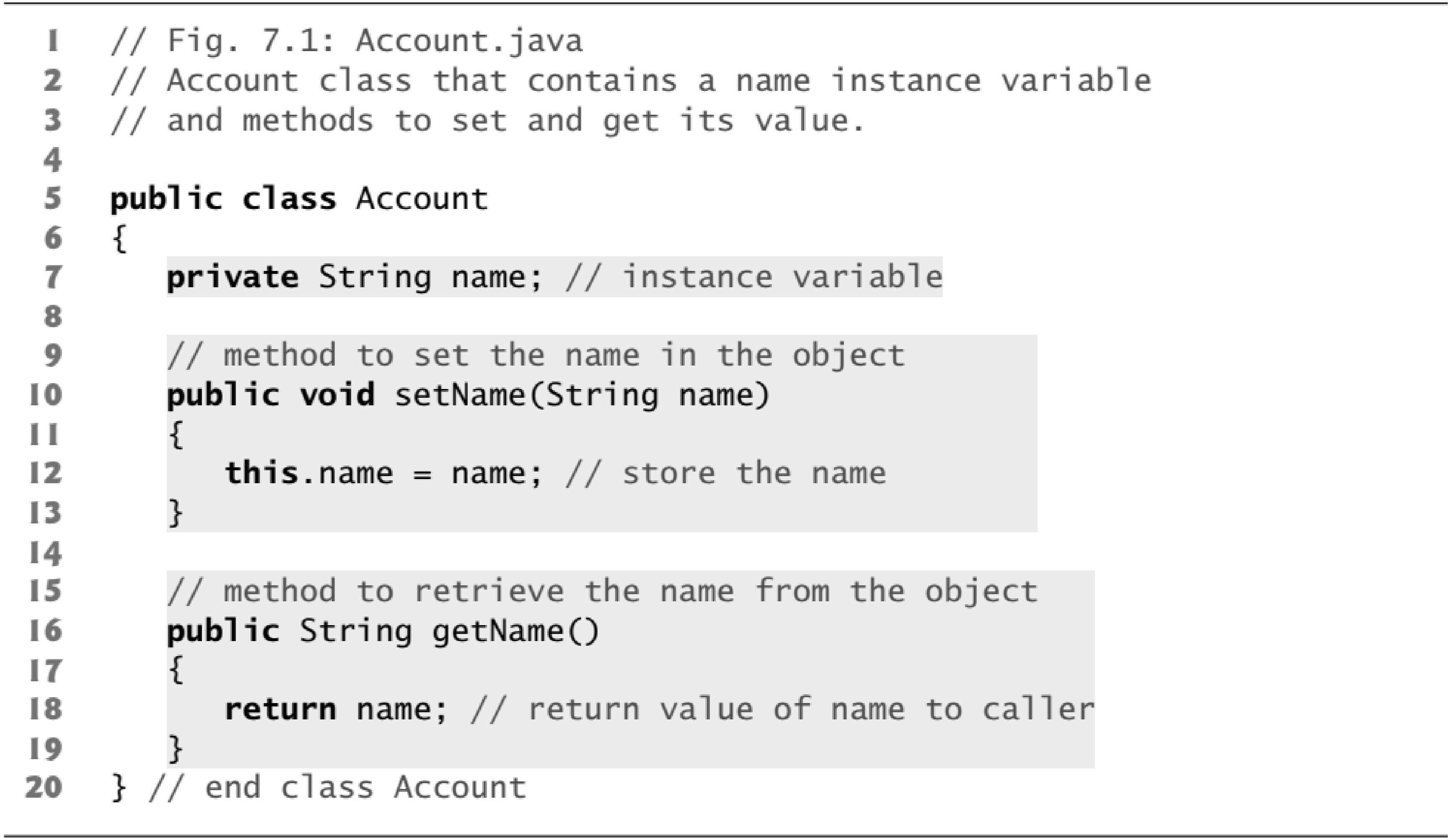
In Java, a method is like a function which is used to expose the behavior of an object.

Methods in classes represent a behavior of an object from the class

Reference: https://www.javatpoint.com/object-and-class-in-java

Declaring classes, instance variables, methods

- Example:

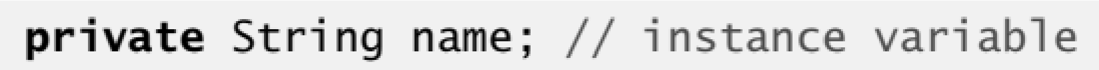


```java
// Fig. 7.1: Account.java
// Account class that contains a name instance variable
// and methods to set and get its value.

public class Account
{
   private String name; // instance variable

   // method to set the name in the object
   public void setName(String name)
   {
      this.name = name; // store the name
   }

   // method to retrieve the name from the object
   public String getName()
   {
      return name; // return value of name to caller
   }
} // end class Account
```

Fig. 7.1 | Account class that contains a name instance variable and methods to *set* and *get* its value.

Declaring classes, instance variables, methods

- *Instance Variable* ***name***



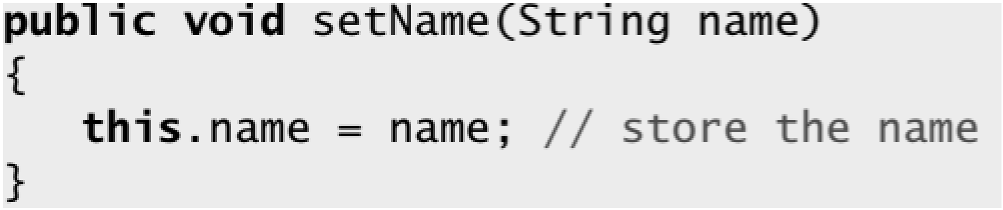
```
private String name; // instance variable
```

  - Instance variables are declared *inside* a class declaration but *outside* the bodies of the class's methods
  - *Access Modifiers* ***public*** *and* ***private***
    - Variables or methods declared with access modifier public are accessible from outside the class in which they're declared.

    - Variables or methods declared with access modifier private are accessible only to methods of the class in which they're declared.

Declaring classes, instance variables, methods

- *Instance Method* ***setName*** *of Class* ***Account***



```
public void setName(String name)
{
   this.name = name; // store the name
}
```

  - indicates that setName receives parameter name of type String—which represents the name that will be passed to the method as an argument.

  - If a method contains a local variable with the *same* name as an instance variable, the method’s body can use the keyword **this** to refer to the shadowed instance variable explicitly

Declaring classes, instance variables, methods

- **AccountTest Class That Creates and Uses an Object of Class Account**

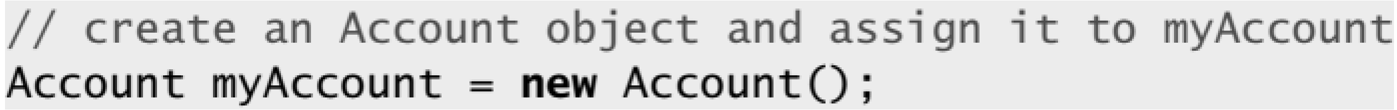
```java
// Fig. 7.2: AccountTest.java
// Creating and manipulating an Account object.
import java.util.Scanner;

public class AccountTest
{
   public static void main(String[] args)
   {
      // create a Scanner object to obtain input from the command window
      Scanner input = new Scanner(System.in);

      // create an Account object and assign it to myAccount
      Account myAccount = new Account();

      // display initial value of name (null)
      System.out.printf("Initial name is: %s%n%n", myAccount.getName());

      // prompt for and read name
      System.out.println("Please enter the name:");
      String theName = input.nextLine(); // read a line of text
      myAccount.setName(theName); // put theName in myAccount
      System.out.println(); // outputs a blank line

      // display the name stored in object myAccount
      System.out.printf("Name in object myAccount is:%n%s%n",
         myAccount.getName());
   }
} // end class AccountTest
```

Output

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Declaring classes, instance variables, methods

- *Instantiating an Object—Keyword **new***



```
// create an Account object and assign it to myAccount
Account myAccount = new Account();
```

  - Keyword **new** creates a new object of the specified class—in this case, Account.
  - The parentheses to the right of Account are *required*.

Declaring classes, instance variables, methods

static Methods

Recall that static methods relate to a class as a whole

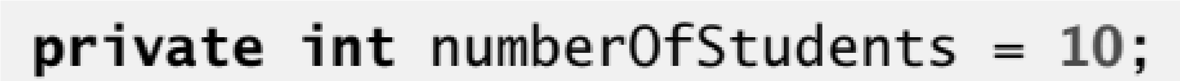
Whereas instance methods are associated with a specific instance (object) of the class and may manipulate the instance variables of that object.

- Default and Explicit Initialization for Instance Variables

Default and Explicit Initialization for Instance Variables

- Recall that local variables are *not* initialized by default.
  - Primitive-type instance variables *are* initialized by default
  - Instance variables of types byte, char, short, int, long, float and double are initialized to 0, and variables of type boolean are initialized to false.

  - You can initialize an instance variable as:



```
private int numberOfStudents = 10;
```

- Constructors, setters, getters, private, public.

Constructors, private, public, static.

- Each class you declare can optionally provide a *constructor* with parameters that can be used to initialize an object of a class when the object is created.

  - It has the same name as its class and is syntactically similar to a method.

  - However, constructors have no explicit return type.

Constructors, private, public, static.

- Example:

```
Public class MyClass {
        Int num; MyClass() {
                num = 100;
        }
}
```

```
public class ConsDemo {
        public static void main(String args[]) {
                MyClass t1 = new MyClass();
                MyClass t2 = new MyClass();
                System.out.println(t1.num + " " + t2.num);
        }
}
```

Output

```
100     100
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

- Parameterized Constructors

```
// A simple constructor.
class MyClass {
        int x;
        MyClass(int i ) {
                x = i;
        }
}
```

```
public class ConsDemo {
        public static void main(String args[]) {
                MyClass t1 = new MyClass( 10 );
                MyClass t2 = new MyClass( 20 );
                System.out.println(t1.x + " " + t2.x);
        }
}
```

Output

```
10      20
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

- If you declare any variable as static, it is known as a static variable.
  - The static variable can be used to refer to the common property of all objects
  - The static variable gets memory only once in the class area at the time of class loading.

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

```
class Student{
  int rollno;//instance variable
  String name;
  static String college ="ITS";//static variable
  Student(int r, String n){
            rollno = r;
            name = n;
  }
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
public class TestStaticVariable1{
            public static void main(String args[]){
                        Student s1 = new Student(111,"Karan");
                        Student s2 = new Student(222,"Aryan");
                        s1.display();
                        s2.display();
             }
}
```

Output

```
111 Karan ITS
222 Aryan ITS
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

- Program of the counter without static variable

```
class Counter{
	int count=0;//will get memory each time when the instance is created
	Counter(){
		count++;//incrementing value
		System.out.println(count);
	}

	public static void main(String args[]){
		//Creating objects
		Counter c1=new Counter();
		Counter c2=new Counter();
		Counter c3=new Counter();
	}
}
```

Output

```
1
1
1
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

- Program of the counter with static variable

```
class Counter{
        static int count=0;
        Counter(){
                count++;//incrementing value
                System.out.println(count);
        }

        public static void main(String args[]){
                //Creating objects
                Counter c1=new Counter();
                Counter c2=new Counter();
                Counter c3=new Counter();
        }
}
```

Output

```
1
2
3
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

- Static Method Example:

```
class Calculate{
		static int cube(int x){
				return x*x*x;
		}

		public static void main(String args[]){
				int result=Calculate.cube(5);
				System.out.println(result);
		}
}
```

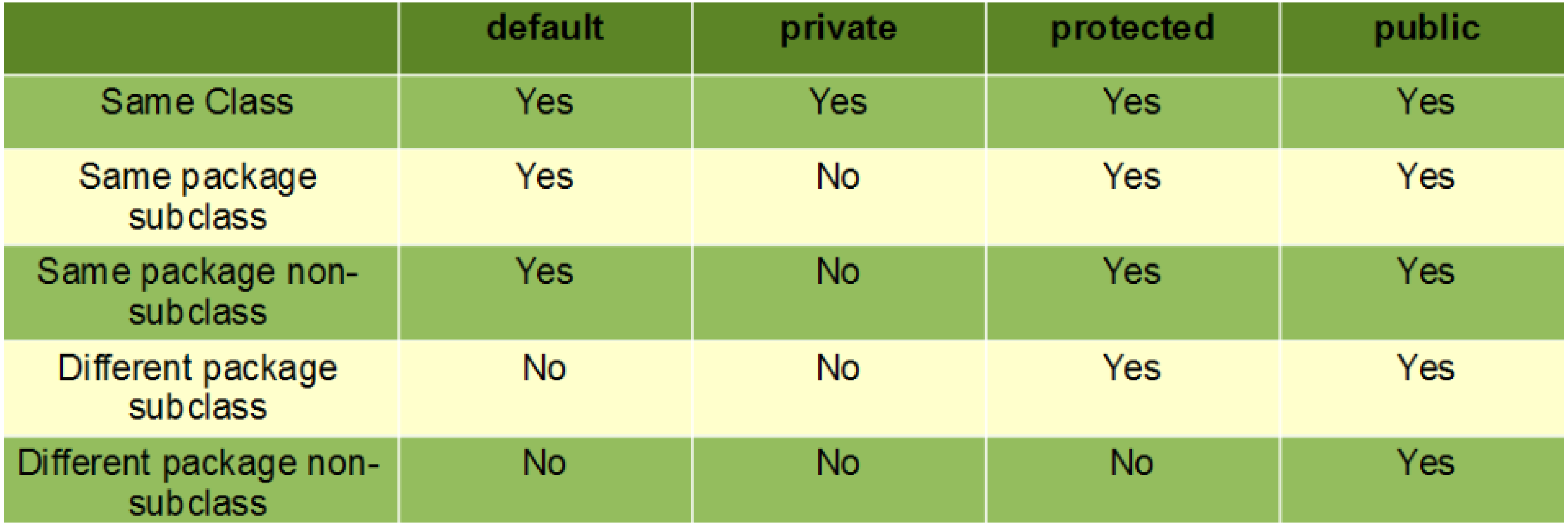
Output

```
125
```

Reference: https://www.javatpoint.com

Constructors, private, public, static.

- Public & Private Modifiers



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Reference: https://www.geeksforgeeks.org/access-modifiers-java/

Constructors, private, public, static.

- Public Modifier

```
package abcpackage;
public class Addition {
    Public int addTwoNumbers(int a, int b){
        return a+b;
    }
}
```

```
package xyzpackage;
import abcpackage.*;
public class Test {
        public static void main(String args[]){
                Addition obj = new Addition();
                obj.addTwoNumbers(10, 21);
        }
}
```

Output

```
31
```

Reference: https://beginnersbook.com/2013/05/java-access-modifiers/

Constructors, private, public, static.

- Private Modifier

```
class ABC{
       private double num = 100;
       private int square(int a){
              return a*a;
       }
}

public class Example{
       public static void main(String args[]){
              ABC obj = new ABC();
              System.out.println(obj.num);
              System.out.println(obj.square(10));
       }
}
```

Output

Compile - time error

Reference: https://beginnersbook.com/2013/05/java-access-modifiers/

- Instantiating and using objects

Instantiating and using objects

- Each of the following statements creates an object and assigns it to a variable:
  - Point originOne = new Point(23, 94);
  - Rectangle rectOne = new Rectangle(originOne, 100, 200);
  - Rectangle rectTwo = new Rectangle(50, 100);

  - Each of these statements has three parts:
    - **Declaration**: The code set in bold are all variable declarations that associate a variable name with an object type.
    - **Instantiation**: The new keyword is a Java operator that creates the object.
    - **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

Reference: https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html

- Implementing a test class

Implementing a test class

- Example 1:

```
class Lamp {
        // stores the value for light
        // true if light is on
        // false if light is off
        boolean isOn;
        // method to turn on the light
        void turnOn() {
                isOn = true;
                System.out.println("Light on? " + isOn);
        }
        // method to turnoff the light
        void turnOff() {
                isOn = false;
                System.out.println("Light on? " + isOn);
        }
}
class Main {
        public static void main(String[] args) {
                // create objects led and halogen
                Lamp led = new Lamp();
                Lamp halogen = new Lamp();
                // turn on the light by calling method turnOn()
                led.turnOn();
                // turn off the light by calling method turnOff()
                halogen.turnOff();
        }
}
```

Output

```
Light on? true
Light on? False
```

Reference: https://www.programiz.com/java-programming/class-objects

Implementing a test class

- Example 2:

```
class Account{
        int acc_no;
        String name;
        float amount;
        void insert(int a,String n,float amt){
                acc_no=a;
                name=n;
                amount=amt;
        }
        void deposit(float amt){
                amount=amount+amt;
                System.out.println(amt+" deposited");
        }
        void withdraw(float amt){
                if(amount<amt){
                        System.out.println("Insufficient Balance");
                }else{
                        amount=amount-amt;
                        System.out.println(amt+" withdrawn");
                }
        }
        //method to check the balance of the account
        void checkBalance(){System.out.println("Balance is: "+amount);}
        //method to display the values of an object
        void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
```

Reference: https://www.javatpoint.com/object-and-class-in-java

Implementing a test class

- Example 2:

```
class TestAccount{
	public static void main(String[] args){
		Account a1=new Account();
		a1.insert(832345,"Ankit",1000);
		a1.display();
		a1.checkBalance();
		a1.deposit(40000);
		a1.checkBalance();
		a1.withdraw(15000);
		a1.checkBalance();
	}
}
```

Output

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Reference: https://www.javatpoint.com/object-and-class-in-java

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية

SAUDI ELECTRONIC UNIVERSITY

2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 11

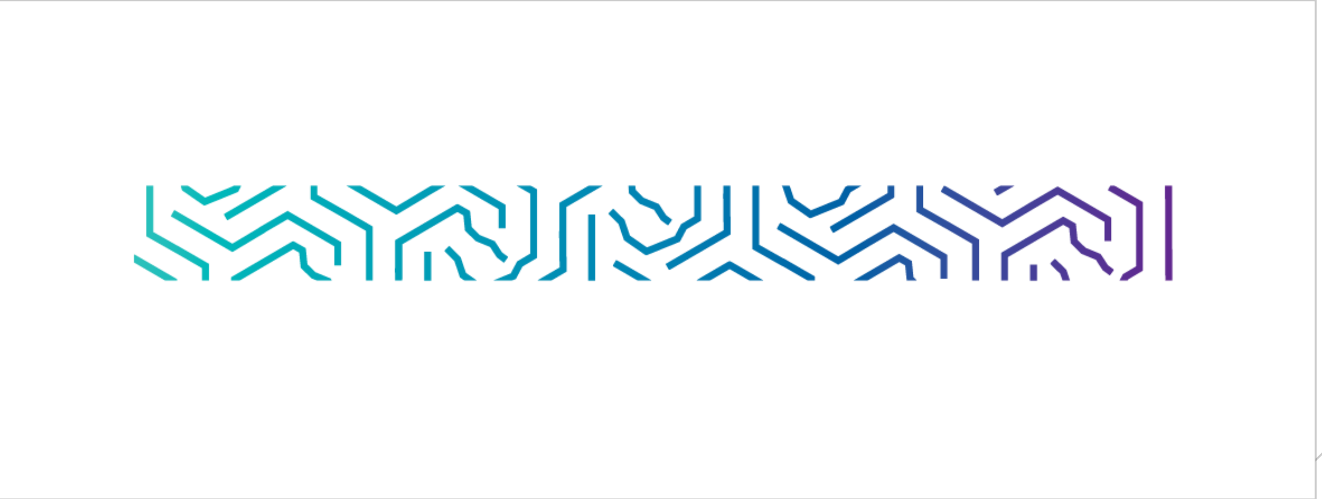
Classes and Objects II

Contents

1. this Reference
2. Overloaded Constructors
3. Default and No-Argument Constructors
4. Composition
5. enum Types
6. Garbage Collection
7. static Class Members & static import
8. final Instance Variables
9. Creating packages

Weekly Learning Outcomes

- Demonstrate encapsulation and data hiding concepts
- Illustrate final and static modifiers



Required Reading

1. Chapter 8 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 7 - (Java The Complete Reference, Eleventh Edition)
2. Final static variables: https://www.geeksforgeeks.org/final-static-variable-java/
3. Garbage Collection in Java: https://beginnersbook.com/2013/04/java-garbage-collection/

- this Reference

this Reference

- The problem if we don't use this keyword by the example given below:

```
class Student{
		int rollno;
		String name;
		float fee;
		Student(int rollno,String name,float fee){
				rollno=rollno;
				name=name;
				fee=fee;
		}
		void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
		public static void main(String args[]){
				Student s1=new Student(111,"ankit",5000f);
				Student s2=new Student(112,"sumit",6000f);
				s1.display();
				s2.display();
}}
```

Output

```
0 null 0.0
0 null 0.0
```

Reference: https://www.javatpoint.com/this-keyword

this Reference

- The problem if we don't use this keyword by the example given below:

```
class Student{
        int rollno;
        String name;
        float fee;
        Student(int rollno,String name,float fee){
                this.rollno=rollno;
                this.name=name;
                this.fee=fee;
        }
        void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
        public static void main(String args[]){
                Student s1=new Student(111,"ankit",5000f);
                Student s2=new Student(112,"sumit",6000f);
                s1.display();
                s2.display();
}}
```

Output

```
111 ankit 5000.0
112 sumit 6000.0
```

Reference: https://www.javatpoint.com/this-keyword

this Reference

- this() : to invoke current class constructor

```
class A{
        A(){
                System.out.println("hello a");
        }
        A(int x){
                this();
                System.out.println(x);
        }
}
class TestThis5{
        public static void main(String args[]){
                A a=new A(10);
        }
}
```

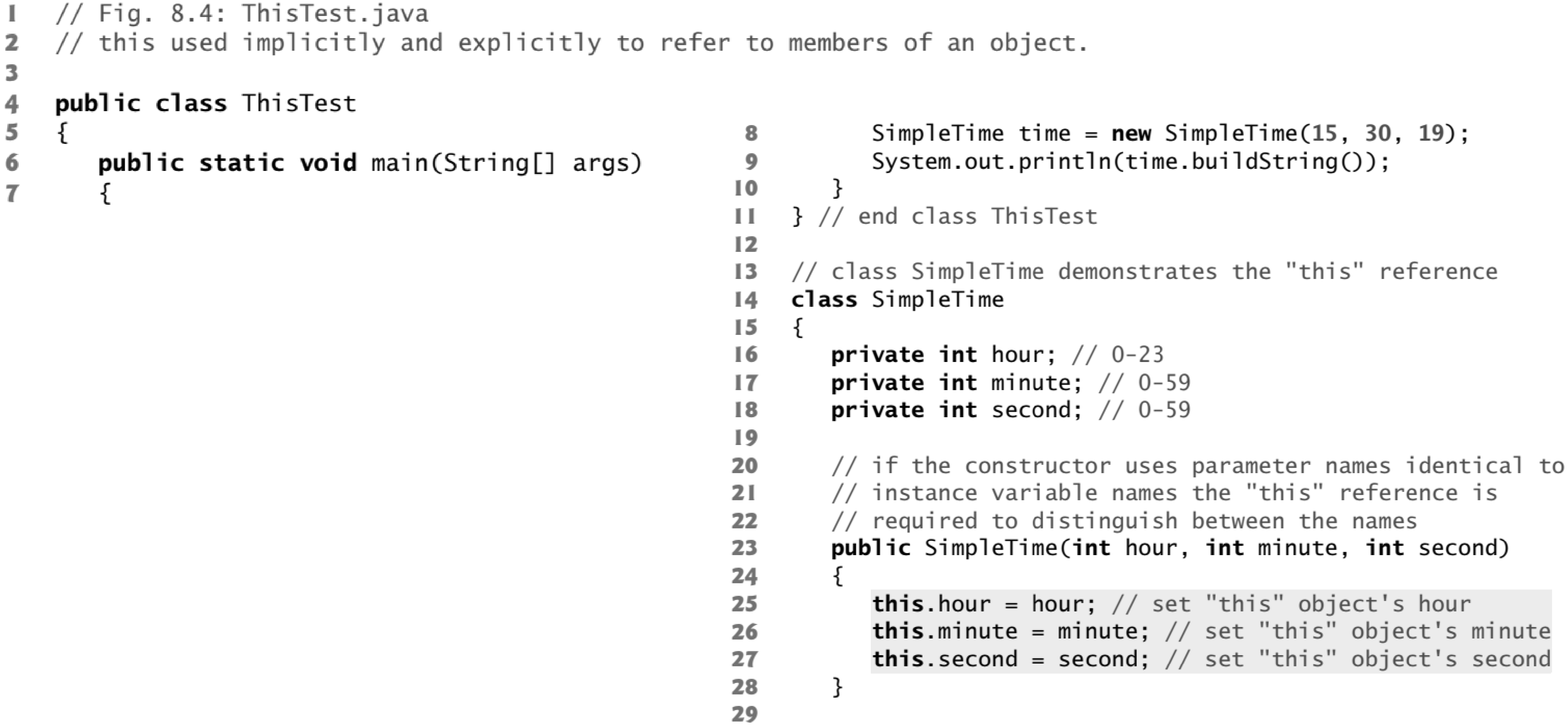
Output

```
hello a
10
```

Reference: https://www.javatpoint.com/this-keyword

this Reference

- Example:



```
// Fig. 8.4: ThisTest.java
// this used implicitly and explicitly to refer to members of an object.

public class ThisTest
{
   public static void main(String[] args)
   {
      SimpleTime time = new SimpleTime(15, 30, 19);
      System.out.println(time.buildString());
   }
} // end class ThisTest

// class SimpleTime demonstrates the "this" reference
class SimpleTime
{
   private int hour; // 0-23
   private int minute; // 0-59
   private int second; // 0-59

   // if the constructor uses parameter names identical to
   // instance variable names the "this" reference is
   // required to distinguish between the names
   public SimpleTime(int hour, int minute, int second)
   {
      this.hour = hour; // set "this" object's hour
      this.minute = minute; // set "this" object's minute
      this.second = second; // set "this" object's second
   }

```

this Reference

```java
   // use explicit and implicit "this" to call toUniversalString
   public String buildString()
   {
      return String.format("%24s: %s%n%24s: %s",
         "this.toUniversalString()", this.toUniversalString(),
         "toUniversalString()", toUniversalString());
   }

   // convert to String in universal-time format (HH:MM:SS)
   public String toUniversalString()
   {
      // "this" is not required here to access instance variables,
      // because method does not have local variables with same
      // names as instance variables
      return String.format("%02d:%02d:%02d",
         this.hour, this.minute, this.second);
   }
} // end class SimpleTime
```

Output

```
this.toUniversalString(): 15:30:19
     toUniversalString(): 15:30:19
```

- Overloaded Constructors

Overloaded Constructors

- Java Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter list.

- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.
- Example of valid constructors for class Account:
  - Account(int a);
  - Account (int a,int b);
  - Account (String a,int b);

Reference: https://www.guru99.com/java-constructors.html

Overloaded Constructors

```
class Demo{
        int value1;
        int value2;
        Demo(){
                value1 = 10;
                value2 = 20;
                System.out.println("Inside 1st Constructor");
        }
        Demo(int a){
                value1 = a;
                System.out.println("Inside 2nd Constructor");
        }
        Demo(int a,int b){
                value1 = a;
                value2 = b;
                System.out.println("Inside 3rd Constructor");
        }
        public void display(){
                System.out.println("Value1 === "+value1);
                System.out.println("Value2 === "+value2);
        }
        public static void main(String args[]){
                Demo d1 = new Demo();
                Demo d2 = new Demo(30);
                Demo d3 = new Demo(30,40);
                d1.display();
                d2.display();
                d3.display();
        }
}
```

Output

```
Inside 1st Constructor
Inside 2nd Constructor
Inside 3rd Constructor
Value1 === 10
Value2 === 20
Value1 === 30
Value2 === 0
Value1 === 30
Value2 === 40
```

Reference: https://www.guru99.com/java-constructors.html

Overloaded Constructors

- **Time Class Case Study**
    - Check the full case study in Chapter 8, pages from 316 to 322

- Default and No-Argument Constructors

Default and No-Argument Constructors

- Every class *must* have at least *one* constructor.
  - If you do not provide any in a class's declaration, the compiler creates a *default constructor* that takes *no* arguments when it's invoked.
- The default constructor initializes the instance variables to the initial values specified in their declarations or to their default values

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.

- Composition

Composition

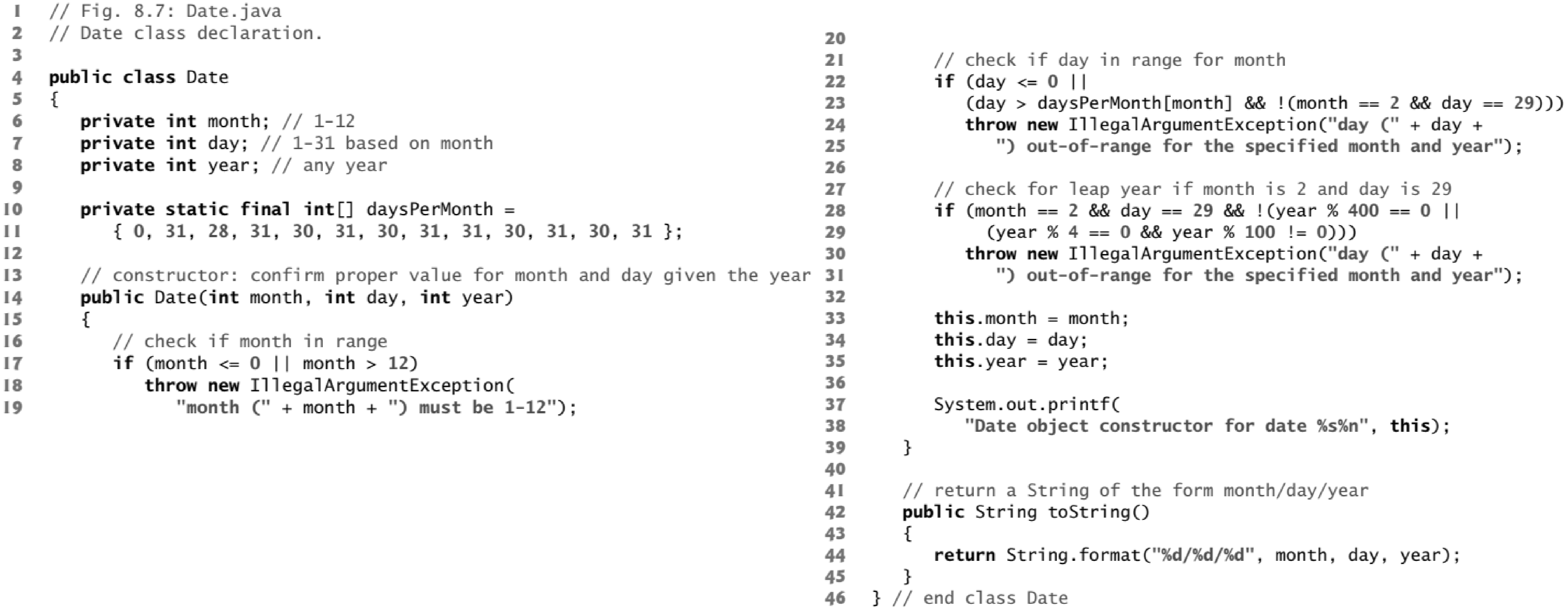
- A class can have references to objects of other classes as members.
    - This is called composition and is sometimes referred to as a *has-a* relationship.

    - Example:
        - An AlarmClock object needs to know the current time *and* the time when it's supposed to sound its alarm
            - it's reasonable to include *two* references to Time objects in an AlarmClock object.

        - A car *has-a* steering wheel, a break pedal and an accelerator pedal.

Composition

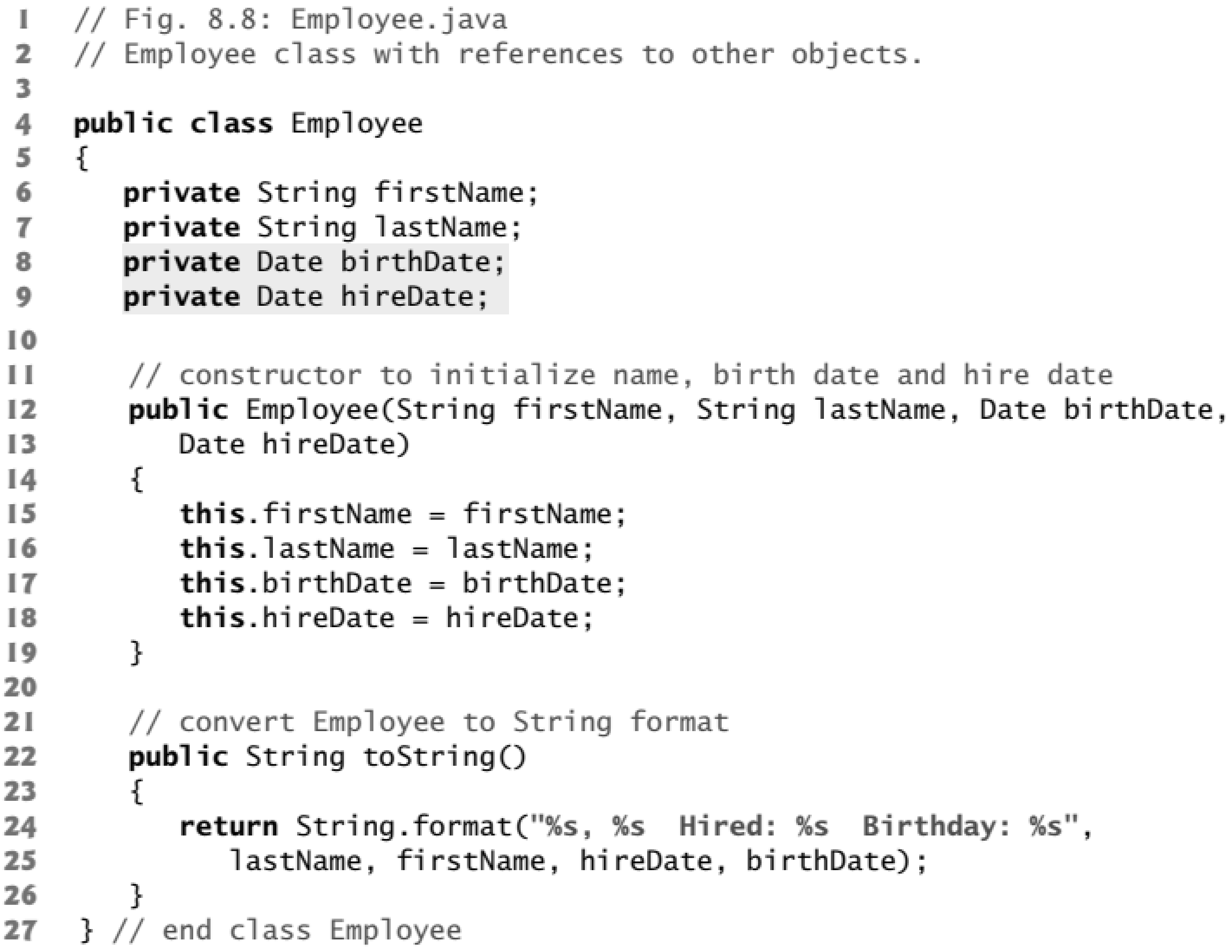
- Example (Class Date):



```java
// Fig. 8.7: Date.java
// Date class declaration.

public class Date
{
   private int month; // 1-12
   private int day; // 1-31 based on month
   private int year; // any year

   private static final int[] daysPerMonth =
      { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

   // constructor: confirm proper value for month and day given the year
   public Date(int month, int day, int year)
   {
      // check if month in range
      if (month <= 0 || month > 12)
         throw new IllegalArgumentException(
            "month (" + month + ") must be 1-12");

      // check if day in range for month
      if (day <= 0 ||
         (day > daysPerMonth[month] && !(month == 2 && day == 29)))
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");

      // check for leap year if month is 2 and day is 29
      if (month == 2 && day == 29 && !(year % 400 == 0 ||
           (year % 4 == 0 && year % 100 != 0)))
         throw new IllegalArgumentException("day (" + day +
            ") out-of-range for the specified month and year");

      this.month = month;
      this.day = day;
      this.year = year;

      System.out.printf(
         "Date object constructor for date %s%n", this);
   }

   // return a String of the form month/day/year
   public String toString()
   {
      return String.format("%d/%d/%d", month, day, year);
   }
} // end class Date
```

Composition

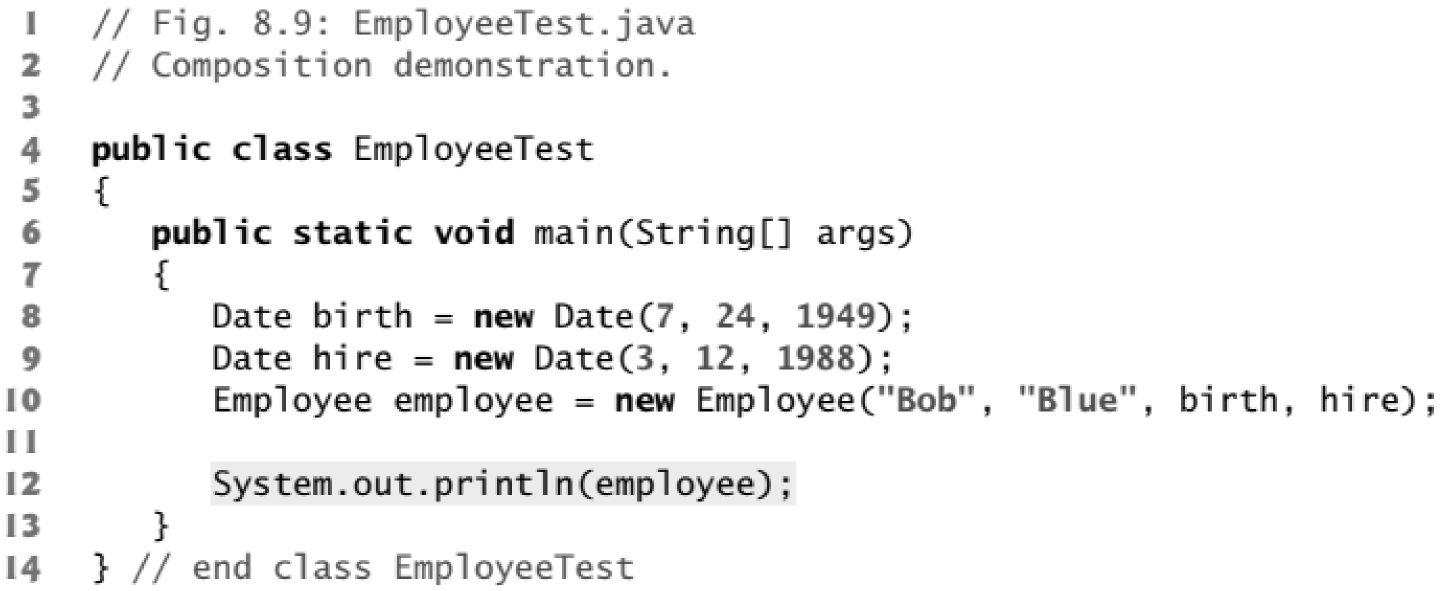
- Class Emplyee



```java
// Fig. 8.8: Employee.java
// Employee class with references to other objects.

public class Employee
{
   private String firstName;
   private String lastName;
   private Date birthDate;
   private Date hireDate;

   // constructor to initialize name, birth date and hire date
   public Employee(String firstName, String lastName, Date birthDate,
      Date hireDate)
   {
      this.firstName = firstName;
      this.lastName = lastName;
      this.birthDate = birthDate;
      this.hireDate = hireDate;
   }

   // convert Employee to String format
   public String toString()
   {
      return String.format("%s, %s  Hired: %s  Birthday: %s",
         lastName, firstName, hireDate, birthDate);
   }
} // end class Employee
```

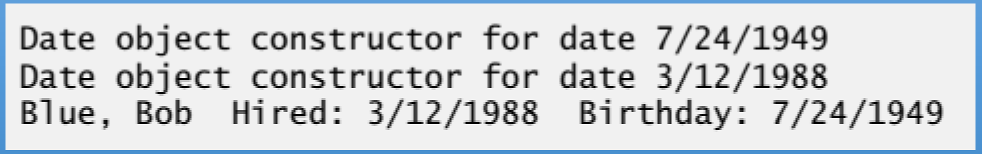
Composition

- Class EmployeeTest



```java
// Fig. 8.9: EmployeeTest.java
// Composition demonstration.

public class EmployeeTest
{
   public static void main(String[] args)
   {
      Date birth = new Date(7, 24, 1949);
      Date hire = new Date(3, 12, 1988);
      Employee employee = new Employee("Bob", "Blue", birth, hire);

      System.out.println(employee);
   }
} // end class EmployeeTest
```

Output



```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

- **enum Types**

enum Types

- It is a special class
  - which defines a set of constants represented as unique identifiers

```
enum Level {
       LOW,
       MEDIUM,
       HIGH
}
```

```
Level myVar = Level.MEDIUM;
```

Reference: https://www.w3schools.com/java/java_enums.asp

enum Types

- Example:

```
class EnumExample1{
        //defining the enum inside the class
        public enum Season { WINTER, SPRING, SUMMER, FALL }
        //main method
        public static void main(String[] args) {
                //traversing the enum
                for (Season s : Season.values())
                        System.out.println(s);
        }
}
```

Output

```
WINTER
SPRING
SUMMER
FALL
```

Reference: https://www.javatpoint.com/enum-in-java

enum Types

- Example:

```
class EnumExample1{
        //defining enum within class
        public enum Season { WINTER, SPRING, SUMMER, FALL }
        //creating the main method
        public static void main(String[] args) {
                //printing all enum
                for (Season s : Season.values()){
                        System.out.println(s);
                }
        System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
        System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
        System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());
        }
}
```

Output

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```

Reference: https://www.javatpoint.com/enum-in-java

- Garbage Collection

Garbage Collection

Object uses system resources, such as memory.

We need a disciplined way to give resources back to the system when they’re no longer needed.

Garbage collection is the process that

Returns memory to system

Java performs this automatically

- object marked for garbage collection if no references to object

Finalizer method

Returns resources to system

Java provides method finalize

- Defined in java.lang.Object
- Receives no parameters
- Returns void

- static Class Members & static import

static Class Members (Revisit) & static import

static keyword

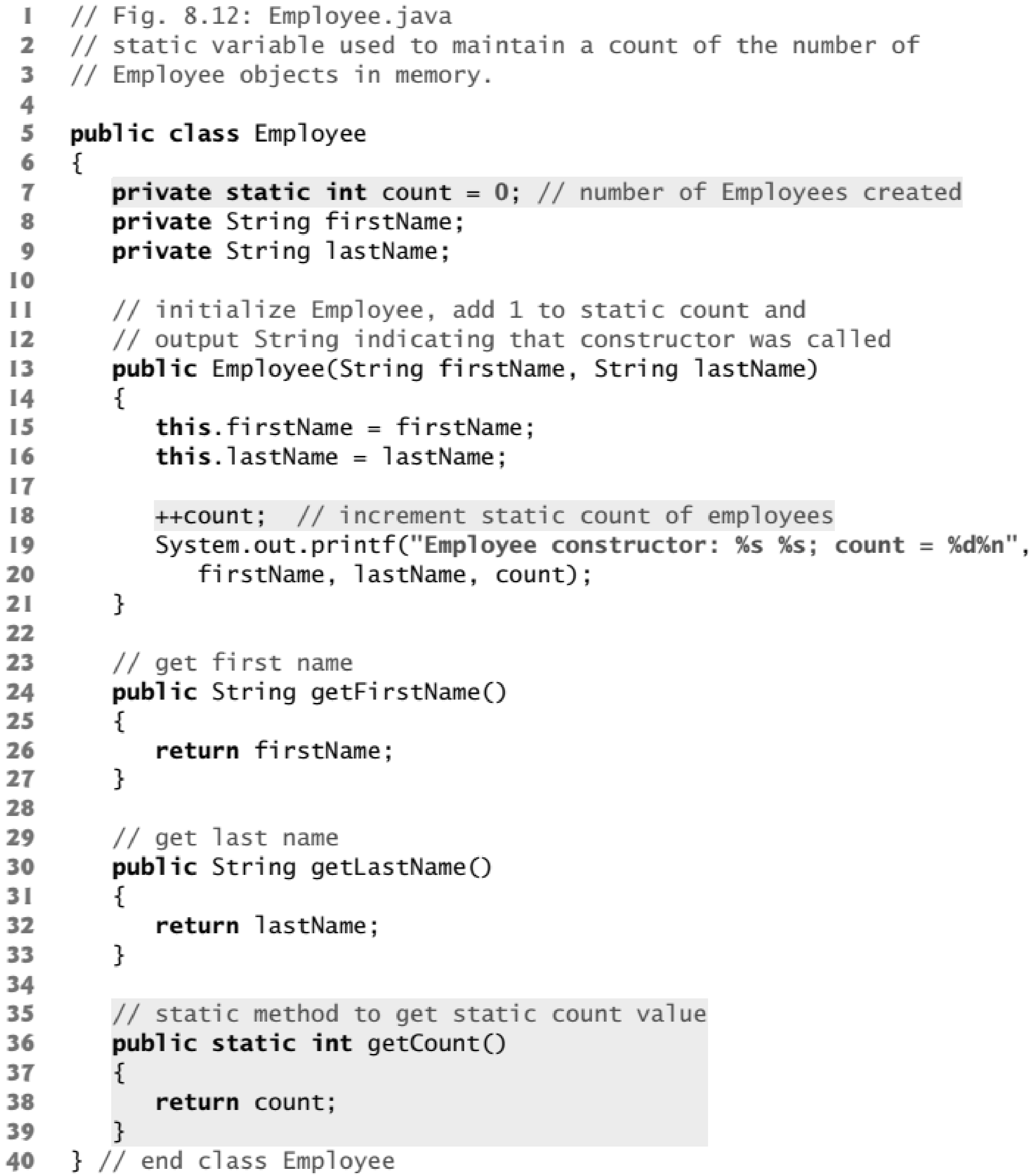
- static class variable
  - Class-wide information
    - All class objects share same data

Access to a class’s public static members

- Qualify the member name with the class name and a dot

static Class Members (Revisit) & static import

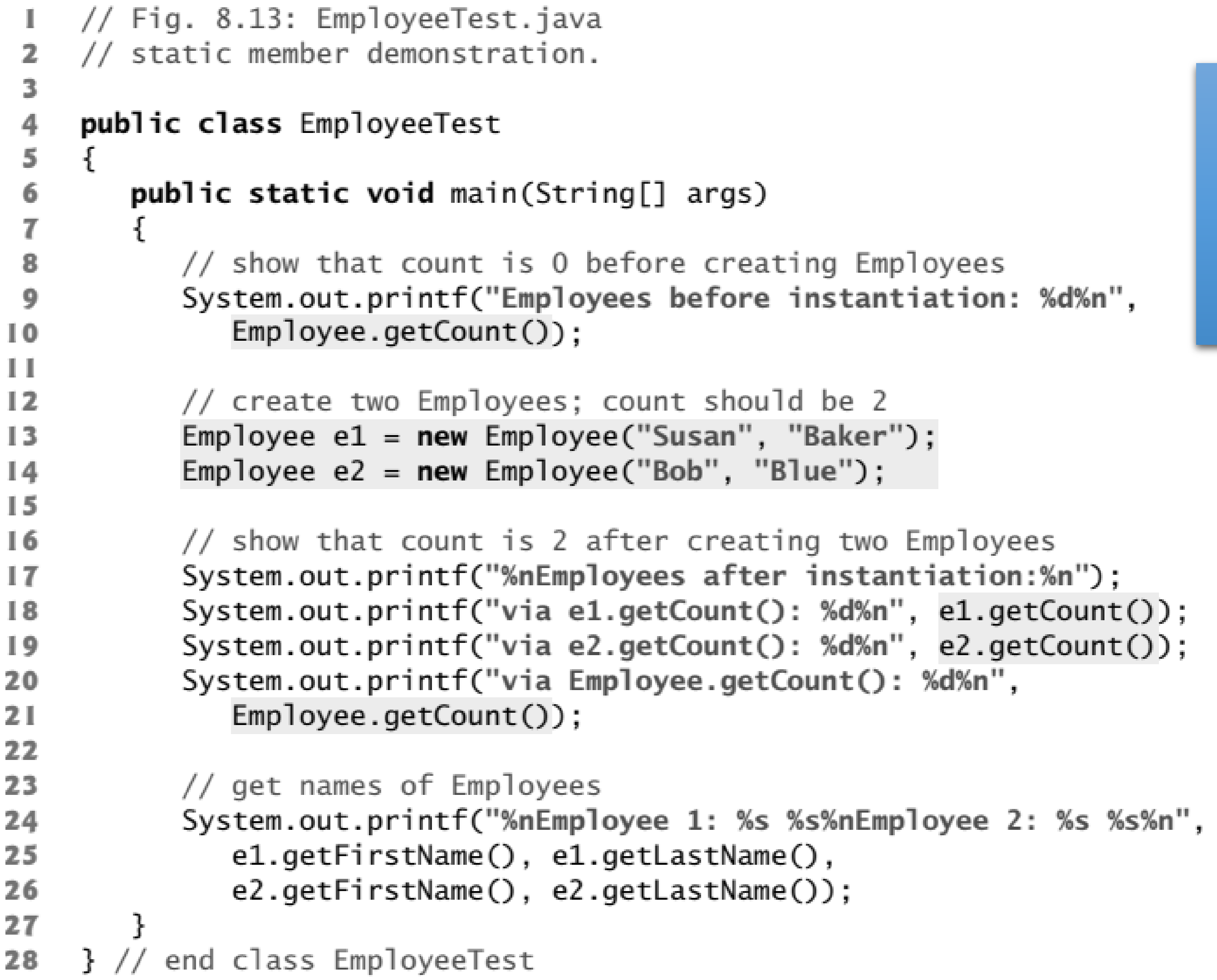
- Example:



```java
// Fig. 8.12: Employee.java
// static variable used to maintain a count of the number of
// Employee objects in memory.

public class Employee
{
   private static int count = 0; // number of Employees created
   private String firstName;
   private String lastName;

   // initialize Employee, add 1 to static count and
   // output String indicating that constructor was called
   public Employee(String firstName, String lastName)
   {
      this.firstName = firstName;
      this.lastName = lastName;

      ++count;  // increment static count of employees
      System.out.printf("Employee constructor: %s %s; count = %d%n",
         firstName, lastName, count);
   }

   // get first name
   public String getFirstName()
   {
      return firstName;
   }

   // get last name
   public String getLastName()
   {
      return lastName;
   }

   // static method to get static count value
   public static int getCount()
   {
      return count;
   }
} // end class Employee
```

static Class Members (Revisit) & static import

- Example:



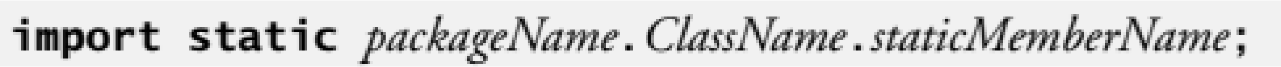
```java
// Fig. 8.13: EmployeeTest.java
// static member demonstration.

public class EmployeeTest
{
   public static void main(String[] args)
   {
      // show that count is 0 before creating Employees
      System.out.printf("Employees before instantiation: %d%n",
         Employee.getCount());

      // create two Employees; count should be 2
      Employee e1 = new Employee("Susan", "Baker");
      Employee e2 = new Employee("Bob", "Blue");

      // show that count is 2 after creating two Employees
      System.out.printf("%nEmployees after instantiation:%n");
      System.out.printf("via e1.getCount(): %d%n", e1.getCount());
      System.out.printf("via e2.getCount(): %d%n", e2.getCount());
      System.out.printf("via Employee.getCount(): %d%n",
         Employee.getCount());

      // get names of Employees
      System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
         e1.getFirstName(), e1.getLastName(),
         e2.getFirstName(), e2.getLastName());
   }
} // end class EmployeeTest
```

Output

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
```

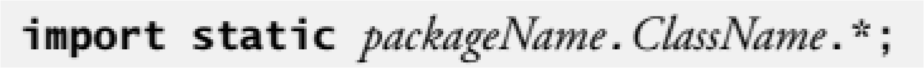
static Class Members (Revisit) & static import

- A **static** import declaration enables you to import the static members of a class or interface so you can access them via their *unqualified names* in your class
  - that is, the class name and a dot (.) are *not* required when using an imported static member.
  - A static import declaration has two forms
    - One that imports a particular static member



```
import static packageName.ClassName.staticMemberName;
```

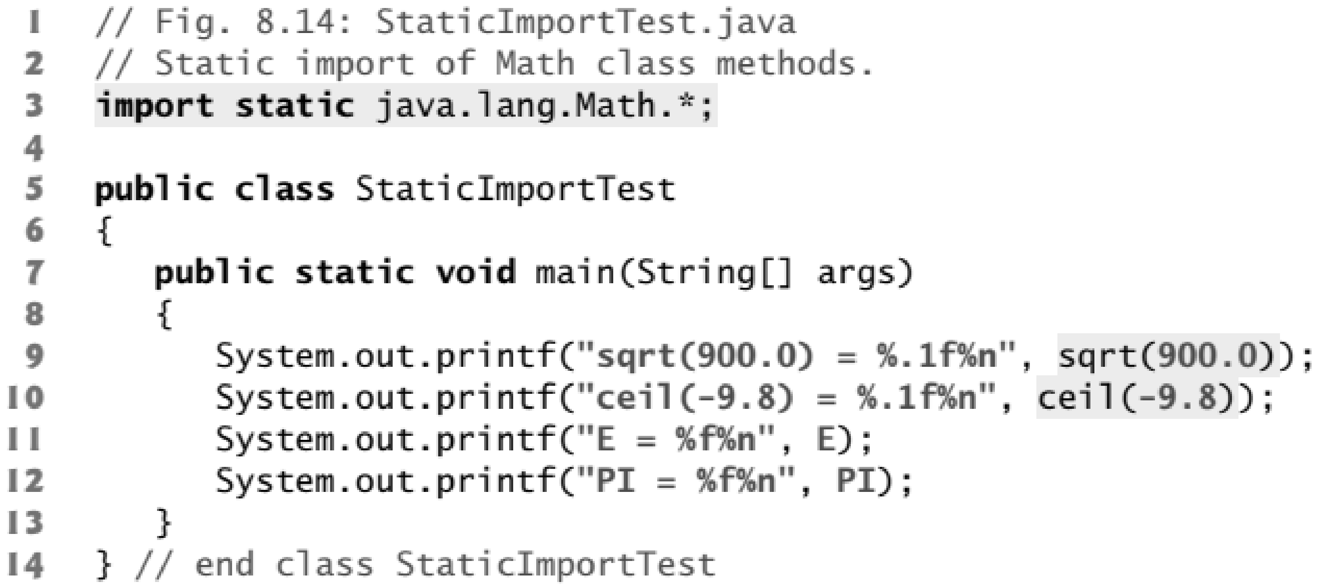
    - One that imports *all* static members of a class



```
import static packageName.ClassName.*;
```

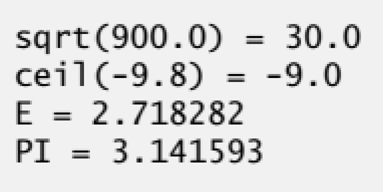
static Class Members (Revisit) & static import

- Example:



```java
// Fig. 8.14: StaticImportTest.java
// Static import of Math class methods.
import static java.lang.Math.*;

public class StaticImportTest
{
   public static void main(String[] args)
   {
      System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
      System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
      System.out.printf("E = %f%n", E);
      System.out.printf("PI = %f%n", PI);
   }
} // end class StaticImportTest
```

Output



```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

- final Instance Variables

final Instance Variables

- `final` keyword
  - Indicates that variable is not modifiable
    - Any attempt to modify `final` variable results in error

```
private final int INCREMENT = 5;
```

    - Declares variable `INCREMENT` as a *constant*
  - Enforces *principle of least privilege*

final Instance Variables

- Example:

```
class Bike9{
        final int speedlimit=90;    //final variable
        void run(){
                speedlimit=400;
        }

        public static void main(String args[]){
                Bike9 obj=new  Bike9();
                obj.run();
        }
}//end of class
```

Output

Compile Time Error

- Creating packages

Creating packages

- Package is a:
  - Group of related classes and interfaces
  - Help manage complexity of application components
  - Facilitate software reuse
  - Provide convention for unique class names
    - Popular package-naming convention
      - Reverse Internet domain name
        - e.g., `com.deitel`

Creating packages

- Create Package pack

```
package pack;
public class A{
        public void msg(){
                System.out.println("Hello");
        }
}
```

- Create Package mypack

```
package mypack;
import pack.*;
 class B{
        public static void main(String args[]){
                A obj = new A();
                obj.msg();
        }
}
```

Output

```
Hello
```

Reference: https://www.javatpoint.com/package

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية

SAUDI ELECTRONIC UNIVERSITY

2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230

Object Oriented Programming

Week 12

Object-Oriented Programming: Inheritance & Polymorphism

Contents

1. Super classes and Subclasses
2. protected Members
3. Relationship Between Super classes and Subclasses
4. Polymorphism Examples
5. Abstract Classes and Methods

Weekly Learning Outcomes

- Compose child classes using the inheritance concept
- Illustrate the polymorphism technique

Required Reading

1. Chapter 9, 10 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 8 - (Java The Complete Reference, Eleventh Edition)
2. Java Inheritance:

https://www.tutorialspoint.com/java/java_inheritance.htm

- Super classes and Subclasses

Super classes and Subclasses

- Inheritance
    - It is a mechanism in which one object acquires all the properties and behaviors of a parent object.
    - Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

    - Features:
        - Software reusability
        - Create new class from existing class
            - Absorb existing class's data and behaviors
            - Enhance with new capabilities

Super classes and Subclasses

- Terms used in Inheritance
  - **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
  - **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
  - **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
  - **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Reference: https://www.javatpoint.com/inheritance-in-java

Super classes and Subclasses

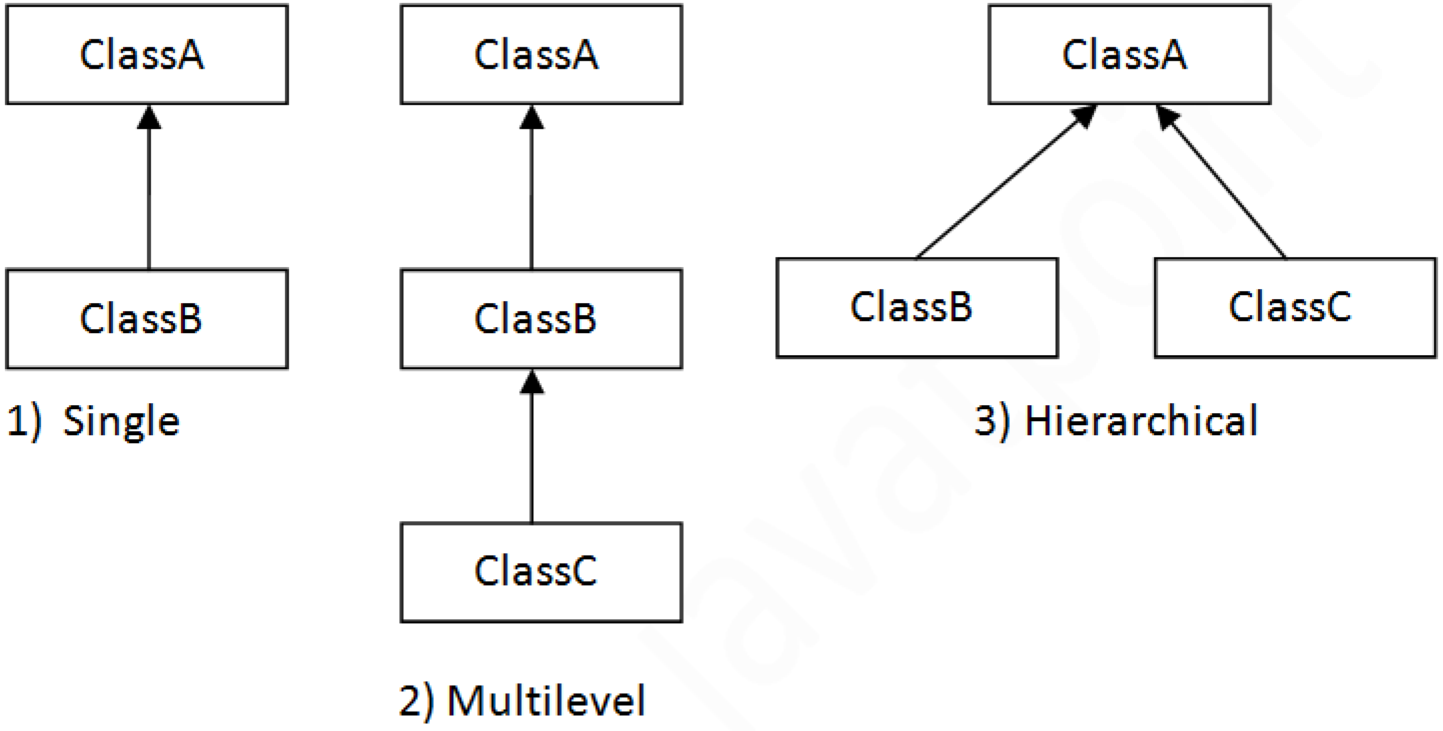
- Subclass extends superclass (referred as base class)
    - Subclass (referred as derived class)
        - More specialized group of objects
        - Behaviors inherited from superclass
            - Can customize
        - Add its own fields and methods
        - Additional behaviors
            - The subclass exhibits
              the behaviors of its superclass and can modify those behaviors

Super classes and Subclasses

- Class hierarchy
  - Direct superclass
    - Inherited explicitly (one level up hierarchy)
  - Indirect superclass
    - Inherited two or more levels up hierarchy
  - Single inheritance
    - Inherits from one superclass
  - Multiple inheritance
    - Inherits from multiple superclasses
      - Java does not support multiple inheritance

Super classes and Subclasses

- Types of inheritance in java



javatpoint

Reference: https://www.javatpoint.com/inheritance-in-java

Super classes and Subclasses

- *is-a* relationship and the *has-a* relationship
  - In an *is-a* relationship:
    - *An object of a subclass can also be treated as an object of its superclass*
      - e.g., a car *is a* vehicle.
        - Vehicle properties/behaviors also car properties/behaviors

  - In *has-a* represents:
    - Composition
    - *An object contains as members references to other objects*
      - e.g., a car *has a* steering wheel

Super classes and Subclasses

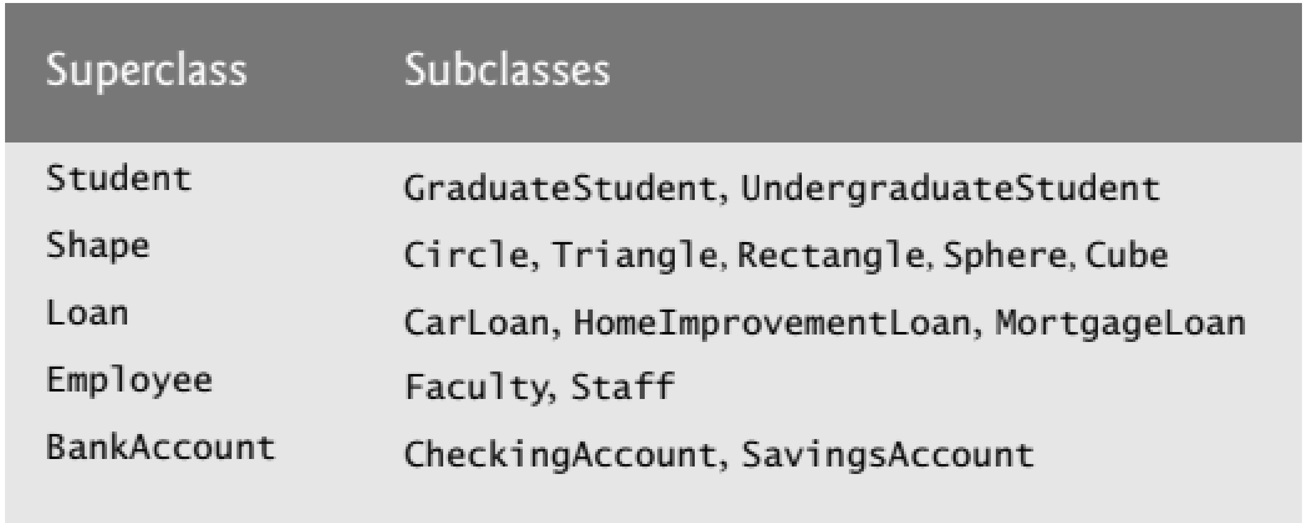
- an object of one class *is an* object of another class as well.
    - For example:
        - A CarLoan *is a* Loan as are HomeImprovementLoans and MortgageLoans.
        - Thus, in Java, class CarLoan can be said to inherit from class Loan.

        - In this context:
            - Class Loan is a superclass
            - Class CarLoan is a subclass
            - A CarLoan *is a* specific type of Loan
            - But it’s incorrect to claim that every Loan *is a* CarLoan—the Loan could be any type of loan.

    - Superclass typically represents larger set of objects than subclasses

Super classes and Subclasses

- Examples:



Superclass	Subclasses
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`, `Sphere`, `Cube`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`BankAccount`	`CheckingAccount`, `SavingsAccount`

Fig. 9.1 | Inheritance examples.

Super classes and Subclasses

- Inheritance hierarchy
  - Inheritance relationships: tree-like hierarchy structure
  - Each class becomes
    - superclass
      - Supply data/behaviors to other classes

    OR

    - subclass
      - Inherit data/behaviors from other classes

Super classes and Subclasses

- *University Community Member Hierarchy*

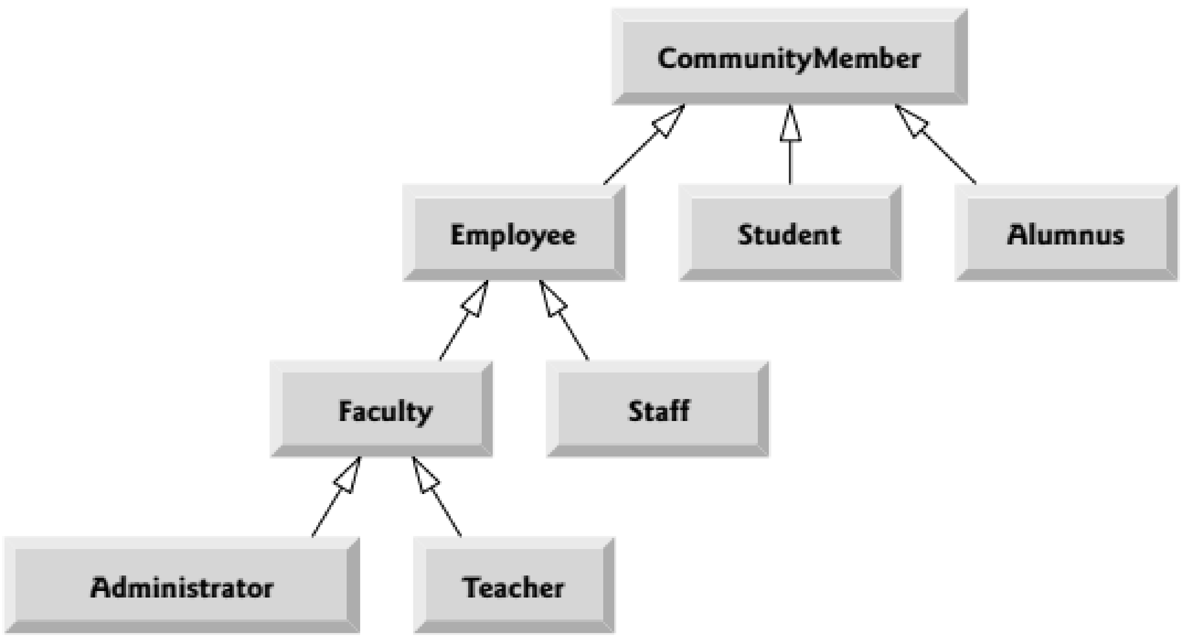


Fig. 9.2 | Inheritance hierarchy UML class diagram for university `CommunityMembers`.

Super classes and Subclasses

- *Shape Hierarchy*

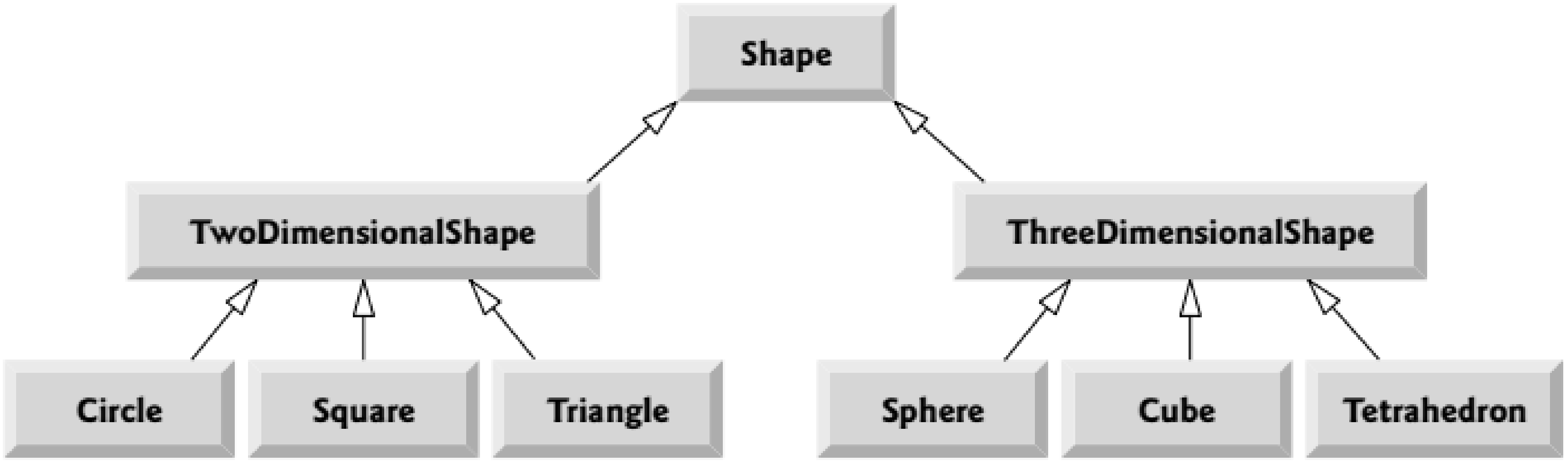


Fig. 9.3 | Inheritance hierarchy UML class diagram for Shapes.

- protected Members

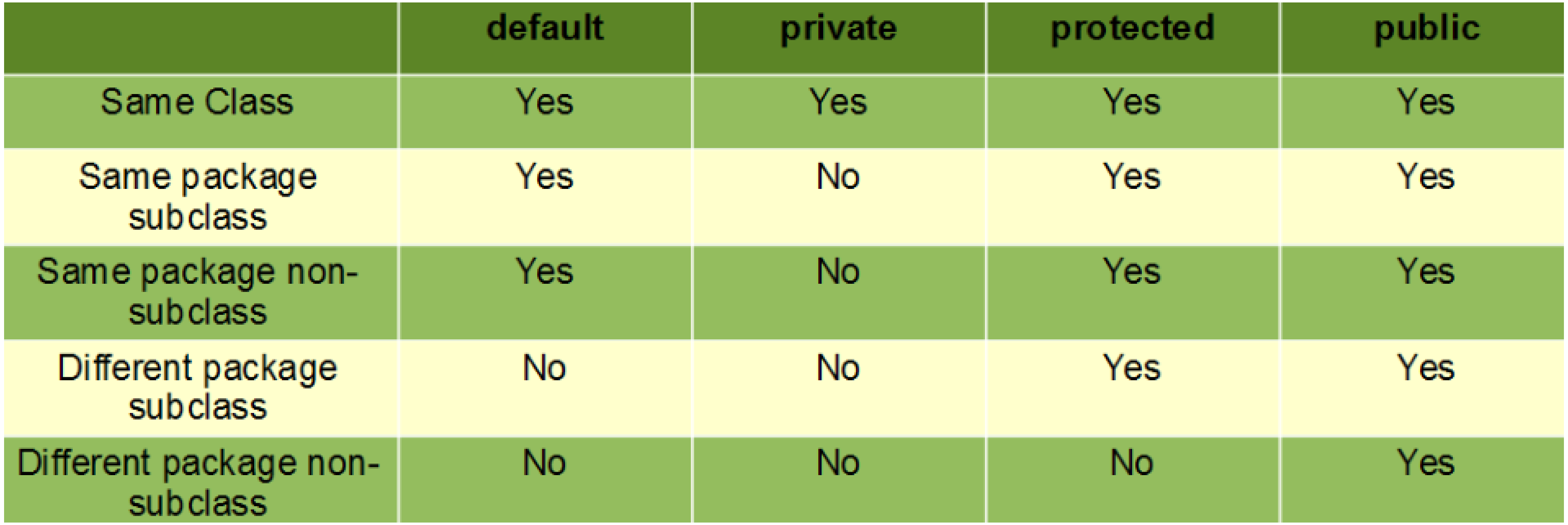
protected Members

- protected access offers:
  - An intermediate level of access between public and private.

  - A superclass’s protected members can be accessed:
    - by members of that superclass
    - by members of its subclasses
    - by members of other classes in the *same package*

  - Subclass access superclass member
    - Keyword `super` and a dot (.)

protected Members



	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Reference: https://www.geeksforgeeks.org/access-modifiers-java/

- Relationship Between Super classes and Subclasses

Relationship Between Super classes and Subclasses

- Superclass and subclass relationship
  - Example: Point/circle inheritance hierarchy
    - Point
      - x-y coordinate pair
    - Circle
      - x-y coordinate pair
      - Radius
      - Area( )

Relationship Between Super classes and Subclasses

- Example:

```
class Employee{
        float salary=40000;
}
class Programmer extends Employee{
        int bonus=10000;

        public static void main(String args[]){
                Programmer p=new Programmer();
                System.out.println("Programmer salary is:"+p.salary);
                System.out.println("Bonus of Programmer is:"+p.bonus);
        }
}
```

Output

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

Reference: https://www.javatpoint.com/inheritance-in-java

Relationship Between Super classes and Subclasses

- Single Inheritance Example

```
class Animal{
        void eat(){
                System.out.println("eating...");
        }
}
class Dog extends Animal{
        void bark(){
                System.out.println("barking...");
        }
}
class TestInheritance{
        public static void main(String args[]){
                Dog d=new Dog();
                d.bark();
                d.eat();
        }
}
```

Output

```
barking...
eating...
```

Reference: https://www.javatpoint.com/inheritance-in-java

Relationship Between Super classes and Subclasses

- Multilevel Inheritance Example

```
class Animal{
        void eat(){    System.out.println("eating...");     }
}
class Dog extends Animal{
        void bark(){   System.out.println("barking...");     }
}
class BabyDog extends Dog{
        void weep(){  System.out.println("weeping...");  }
}
class TestInheritance2{
        public static void main(String args[]){
                BabyDog d=new BabyDog();
                d.weep();
                d.bark();
                d.eat();
        }
}
```

Output

```
weeping...
barking...
eating...
```

Reference: https://www.javatpoint.com/inheritance-in-java

Relationship Between Super classes and Subclasses

- Hierarchical Inheritance Example

```
class Animal{
        void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
        void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
        void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
        public static void main(String args[]){
                Cat c=new Cat();
                c.meow();
                c.eat();
                //c.bark();//C.T.Error
        }
}
```

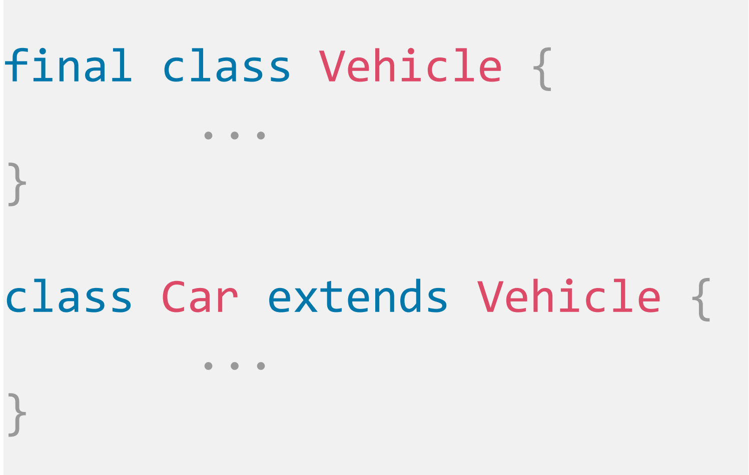
Output

```
meowing...
eating...
```

Reference: https://www.javatpoint.com/inheritance-in-java

Relationship Between Super classes and Subclasses

- The final Keyword
  - If you don't want other classes to inherit from a class, use the final keyword:



```
final class Vehicle {
        ...
}

class Car extends Vehicle {
        ...
}
```

```
Output

Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
                   ^
1 error)
```

Reference: https://www.w3schools.com/java/java_inheritance.asp

Relationship Between Super classes and Subclasses

- **Example for Super Keyword in Java:**

```
class Superclass  {
                  int  i =20;
                  void  display()  {
                                   System.out.println(“Superclass display method”);
                   }
}
class Subclass extends Superclass  {
                  int  i = 100;
                  void  display()   {
                                   super.display();
                                    System.out.println(“Subclass display method”);
                                    System.out.println(“ i value =”+i);
                                    System.out.println(“superclass i value =”+super.i);
                   }
}
class SuperUse {
                   public  static  void  main(String args[])  {
                                   Subclass obj = new  Subclass();
                                    obj.display();
                  }
}
```

Output

```
Superclass display method
Subclass display method
i value =100
superclass i value =20
```

Reference: https://www.mygreatlearning.com/blog/inheritance-in-java/

Relationship Between Super classes and Subclasses

- **Example for protected members:**

```
public class Shape {
        protected double height;
        protected double width;
        private double test;
        public void setValues(double height, double width) {
                this.height = height;
                this.width = width;
        }
}

public class Rectangle extends Shape {
        public double getArea() {
                // test = 10.2;        // Error cannot access private variable
                return height * width; //accessing protected members
        }
 }
```

Reference: https://www.mygreatlearning.com/blog/inheritance-in-java/

Relationship Between Super classes and Subclasses

- **Example for protected members:**

```
 public class Triangle extends Shape {
         public double getArea() {
                 return height * width / 2;
          }
 }


public class TestProgram {
        public static void main(String[] args) {
                Rectangle rectangle = new Rectangle();
                Triangle triangle = new Triangle();
                rectangle.setValues(5,4);
                triangle.setValues(5,10);
                System.out.println("Area of rectangle : " + rectangle.getArea());
                System.out.println("Area of triangle : " + triangle.getArea());
        }
}
```

Output

```
Area of rectangle : 20.0
Area of triangle : 25.0
```

Reference: https://www.mygreatlearning.com/blog/inheritance-in-java/

- Polymorphism Examples

Polymorphism Examples

- Polymorphism
  - means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
  - **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.
  - Treat objects in same class hierarchy as if all superclass
  - Abstract class
    - Common functionality
  - Makes programs extensible
    - New classes added easily, can still be processed
- In our examples
  - Use abstract superclass `Shape`
    - Defines common interface (functionality)
    - `Point`, `Circle` and `Cylinder` inherit from `Shape`
  - Class `Employee` for a natural example

Polymorphism Examples

- Example:

```
class Shapes {
                public void area() {
                                System.out.println("The formula for area of ");
                }
}
class Triangle extends Shapes {
                public void area() {
                                System.out.println("Triangle is ½ * base * height ");
                }
}
class Circle extends Shapes {
                public void area() {
                                System.out.println("Circle is 3.14 * radius * radius ");
                }
}
class Main {
                public static void main(String[] args) {
                                Shapes myShape = new Shapes();  // Create a Shapes object
                                Shapes myTriangle = new Triangle();  // Create a Triangle object
                                Shapes myCircle = new Circle();  // Create a Circle object
                                myShape.area();
                                myTriangle.area();
                                myShape.area();
                                myCircle.area();
                }
}
```

Output

```
The formula for area of Triangle is ½ * base * height
The formula for area of Circle is 3.14 * radius * radius
```

Reference: https://www.mygreatlearning.com/blog/polymorphism-in-java/

Polymorphism Examples

Polymorphism in Java can be performed by two different methods:

Method Overloading

- Defined as a process that can create multiple methods of the same name in the same class, and all the methods work in different ways.
- Method overloading occurs when there is more than one method of the same name in the class

Method Overriding

- Defined as a process when the subclass or a child class has the same method as declared in the parent class.

Reference: https://www.mygreatlearning.com/blog/polymorphism-in-java/

Polymorphism Examples

- Method Overriding Example:

```
class Vehicle{
        void run(){
                System.out.println("Vehicle is moving");
        }
}

class Car2 extends Vehicle{
        void run(){
                System.out.println("car is running safely");
        }


        public static void main(String args[]){
                Car2 obj = new Car2();
                obj.run();//calling method
        }
}
```

Reference: https://www.mygreatlearning.com/blog/polymorphism-in-java/

Polymorphism Examples

- Overriding is done by using a reference variable of the superclass.
  - Which method to be called is determined based on the object which is being referred to by the reference variable.
  - This is also known as **Upcasting**.

```
class A{}

class B extends A{}

A a=new B(); //upcasting
```

Reference: https://www.mygreatlearning.com/blog/polymorphism-in-java/

Polymorphism Examples

- **Example:**

```
class Animal {
        void eat(){
                System.out.println("Animals Eat");
        }
}
class herbivores extends Animal{
        void eat(){
                System.out.println("Herbivores Eat Plants");
        }
 }
class omnivores extends Animal{
        void eat(){
                System.out.println("Omnivores Eat Plants and meat");
        }
 }
class carnivores extends Animal{
        void eat(){
                System.out.println("Carnivores Eat meat");
        }
 }
```

```
class main{
        public static void main(String args[]){
                Animal A = new Animal();
                Animal h = new herbivores(); //upcasting
                Animal o = new omnivores(); //upcasting
                Animal c = new carnivores(); //upcasting
                A.eat();
                h.eat();
                o.eat();
                c.eat();
        }
}
```

Output

```
Animals eat
Herbivores Eat Plants
Omnivores Eat Plants and meat
Carnivores Eat meat
```

Reference: https://www.mygreatlearning.com/blog/polymorphism-in-java/

- Abstract Classes and Methods

Abstract Classes and Methods

Abstract class:

is a restricted class that cannot be used to create objects

to access it, it must be inherited from another class.

Abstract method:

can only be used in an abstract class, and it does not have a body.

The body is provided by the subclass (inherited from).

Reference: https://www.w3schools.com/java/java_abstract.asp

Abstract Classes and Methods

- **Example:**

```
abstract class Animal {
        public abstract void animalSound();
        public void sleep() {
                System.out.println("Zzz");
        }
}
```

- **If you try to execute:**

Animal myObj = new Animal(); // will generate an error

Reference: https://www.w3schools.com/java/java_abstract.asp

Abstract Classes and Methods

- **Example:**

```
abstract class Animal {
        public abstract void animalSound();
        public void sleep() {
                System.out.println("Zzz");
        }
}

class Pig extends Animal {
        public void animalSound() {
                System.out.println("The pig says: wee wee");
        }
}

class Main {
        public static void main(String[] args) {
                Pig myPig = new Pig();
                myPig.animalSound(); myPig.sleep();
        }
}
```

Output

```
The pig says: wee wee
Zzz
```

Reference: https://www.w3schools.com/java/java_abstract.asp

Abstract Classes and Methods

- **An interface:**
  - **is a completely "abstract class" that is used to group related methods with empty bodies.**

  - To access the interface methods, the interface must be "implemented“.

  - The body of the interface method is provided by the "implement" class

Reference: https://www.w3schools.com/java/java_abstract.asp

Abstract Classes and Methods

- **Interface Example:**

```
interface Animal {
        public void animalSound();
        public void sleep();
}

class Pig implements Animal {
        public void animalSound() {
                System.out.println("The pig says: wee wee");
        }
        public void sleep() {
                System.out.println("Zzz");
        }
}

class Main {
        public static void main(String[] args) {
                Pig myPig = new Pig();
                myPig.animalSound(); myPig.sleep();
        }
}
```

Output

```
The pig says: wee wee
Zzz
```

Reference: https://www.w3schools.com/java/java_abstract.asp

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 13

Strings and Characters

Contents

1. Fundamentals of strings and characters
2. Class String, Class Character
3. Tokenizing Strings
4. Regular Expressions

Weekly Learning Outcomes

- Define String and Character Classes
- Recognize the immutable classes

Required Reading

1. Chapter 14 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 17 - (Java The Complete Reference, Eleventh Edition)
2. String Class Tutorial: https://www.tutorialspoint.com/java/java_strings.htm
3. Immutable Objects in Java: https://www.baeldung.com/java-immutable-object

- Fundamentals of strings and characters

Fundamentals of strings and characters

Characters:

- The fundamental building blocks of Java source programs.
- A character literal is an integer value represented as a character in single quotes.
- The value of a character literal is the integer value of the character in the Unicode character set.

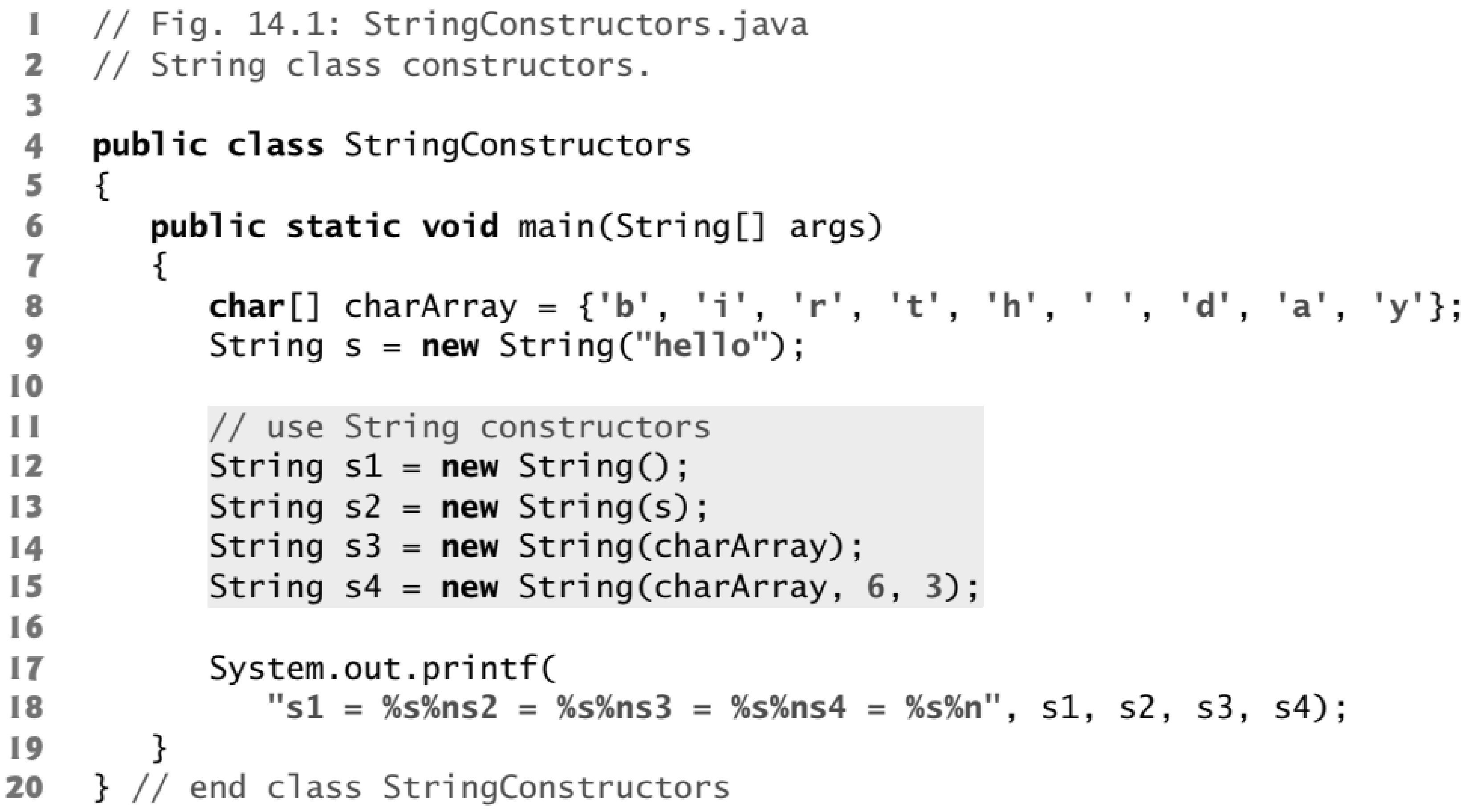
Class String:

- used to represent strings in Java.

- Class String, Class Character

Class String, Class Character

- Class String provides constructors for initializing String objects in a variety of ways.



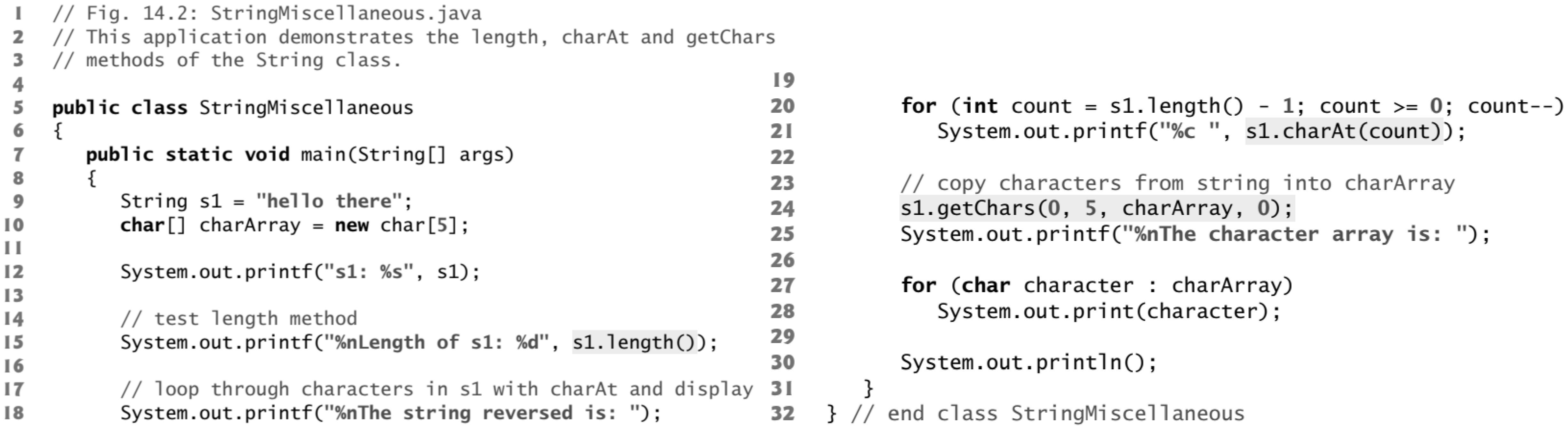
```java
// Fig. 14.1: StringConstructors.java
// String class constructors.

public class StringConstructors
{
   public static void main(String[] args)
   {
      char[] charArray = {'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y'};
      String s = new String("hello");

      // use String constructors
      String s1 = new String();
      String s2 = new String(s);
      String s3 = new String(charArray);
      String s4 = new String(charArray, 6, 3);

      System.out.printf(
         "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n", s1, s2, s3, s4);
   }
} // end class StringConstructors
```

Output

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

Class String, Class Character

- **String Methods length, charAt and getChars**



```
// Fig. 14.2: StringMiscellaneous.java
// This application demonstrates the length, charAt and getChars
// methods of the String class.

public class StringMiscellaneous
{
   public static void main(String[] args)
   {
      String s1 = "hello there";
      char[] charArray = new char[5];

      System.out.printf("s1: %s", s1);

      // test length method
      System.out.printf("%nLength of s1: %d", s1.length());

      // loop through characters in s1 with charAt and display
      System.out.printf("%nThe string reversed is: ");

      for (int count = s1.length() - 1; count >= 0; count--)
         System.out.printf("%c ", s1.charAt(count));

      // copy characters from string into charArray
      s1.getChars(0, 5, charArray, 0);
      System.out.printf("%nThe character array is: ");

      for (char character : charArray)
         System.out.print(character);

      System.out.println();
   }
} // end class StringMiscellaneous
```

Output

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h
The character array is: hello
```

Class String, Class Character

- **Comparing Strings**
  - **Java String class compareTo()** method
    - compares the given string with the current string lexicographically.

1.if s1 > s2, it returns positive number
2.if s1 < s2, it returns negative number
3.if s1 == s2, it returns 0

```java
String myStr1 = "Hello";
String myStr2 = "Hello";
System.out.println(myStr1.compareTo(myStr2));
```

Output

Returns 0 because they are equal

Class String, Class Character

- **Class Character**

```java
// Fig. 14.15: StaticCharMethods.java
// Character static methods for testing characters and converting case.
import java.util.Scanner;

public class StaticCharMethods
{
   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in); // create scanner
      System.out.println("Enter a character and press Enter");
      String input = scanner.next();
      char c = input.charAt(0); // get input character

      // display character info
      System.out.printf("is defined: %b%n", Character.isDefined(c));
      System.out.printf("is digit: %b%n", Character.isDigit(c));
      System.out.printf("is first character in a Java identifier: %b%n",
         Character.isJavaIdentifierStart(c));
      System.out.printf("is part of a Java identifier: %b%n",
         Character.isJavaIdentifierPart(c));
      System.out.printf("is letter: %b%n", Character.isLetter(c));
      System.out.printf(
         "is letter or digit: %b%n", Character.isLetterOrDigit(c));
      System.out.printf(
         "is lower case: %b%n", Character.isLowerCase(c));
      System.out.printf(
         "is upper case: %b%n", Character.isUpperCase(c));
      System.out.printf(
         "to upper case: %s%n", Character.toUpperCase(c));
      System.out.printf(
         "to lower case: %s%n", Character.toLowerCase(c));
   }
} // end class StaticCharMethods
```

Output

```
Enter a character and press Enter
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

Class String, Class Character

- **Check the book for more examples on String and Character:**
  - **String Class Methods:**
    - ***String*** *Method* ***equals***
    - ***String*** *Method* ***equalsIgnoreCase***
    - ***String*** *Method* ***regionMatches***
    - ***String*** *Methods* ***startsWith*** *and* ***endsWith***
    - **String Method valueOf**
  - **Locating Characters and Substrings in Strings**
  - **Extracting Substrings from Strings**
  - **Concatenating Strings**

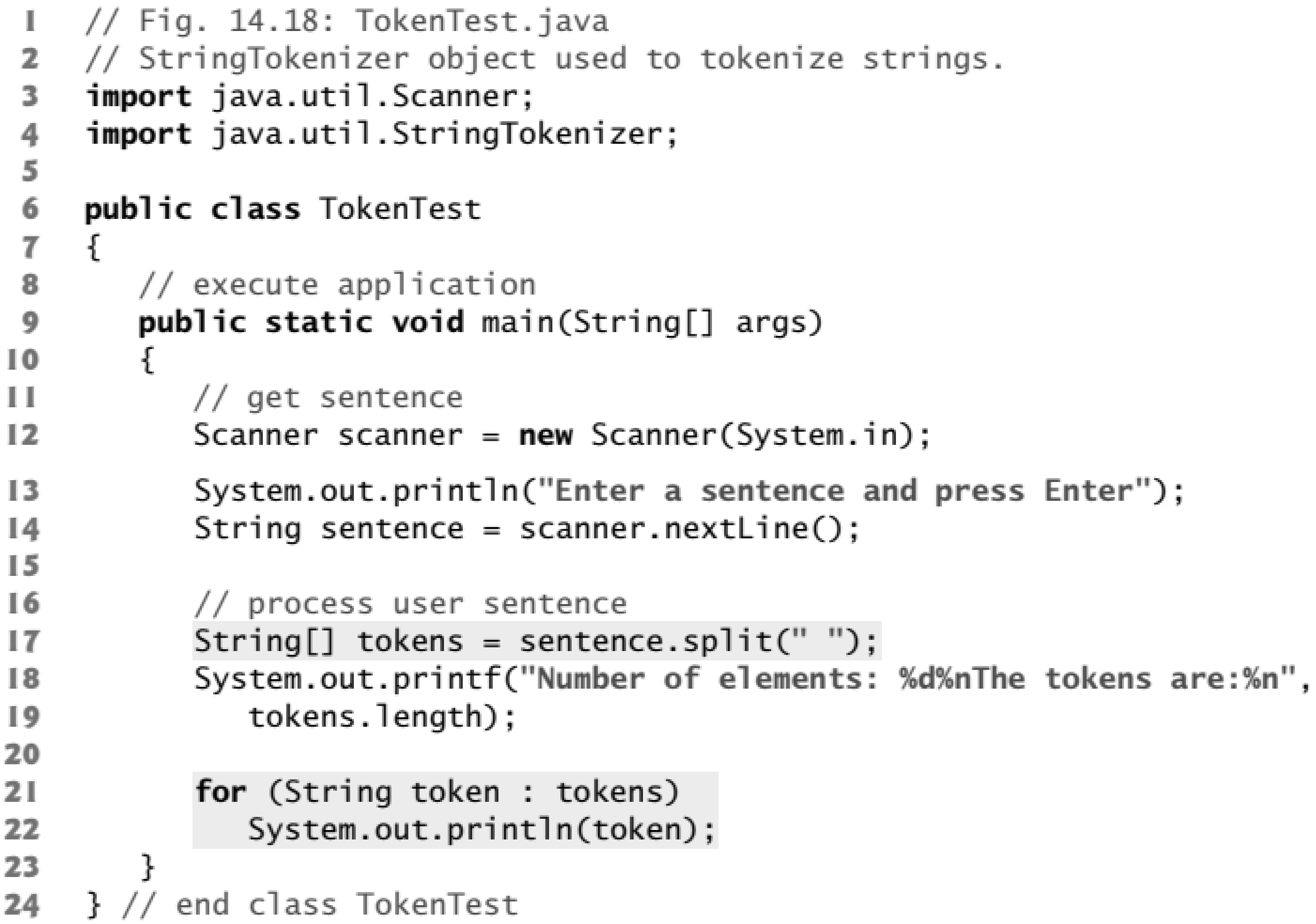
- Tokenizing Strings

Tokenizing Strings

- Tokenizer
  - Partition `String` into individual substrings
  - Use *delimiter*
  - Java offers `java.util.StringTokenizer`

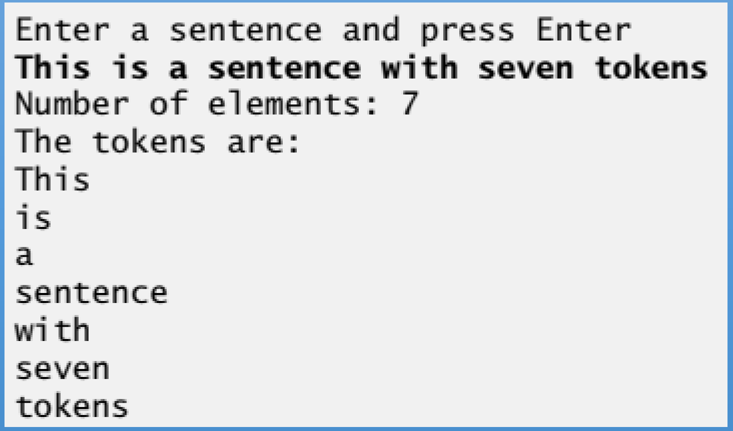
Tokenizing Strings

- Example:



```
// Fig. 14.18: TokenTest.java
// StringTokenizer object used to tokenize strings.
import java.util.Scanner;
import java.util.StringTokenizer;

public class TokenTest
{
   // execute application
   public static void main(String[] args)
   {
      // get sentence
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter a sentence and press Enter");
      String sentence = scanner.nextLine();

      // process user sentence
      String[] tokens = sentence.split(" ");
      System.out.printf("Number of elements: %d%nThe tokens are:%n",
         tokens.length);

      for (String token : tokens)
         System.out.println(token);
   }
} // end class TokenTest
```

Output

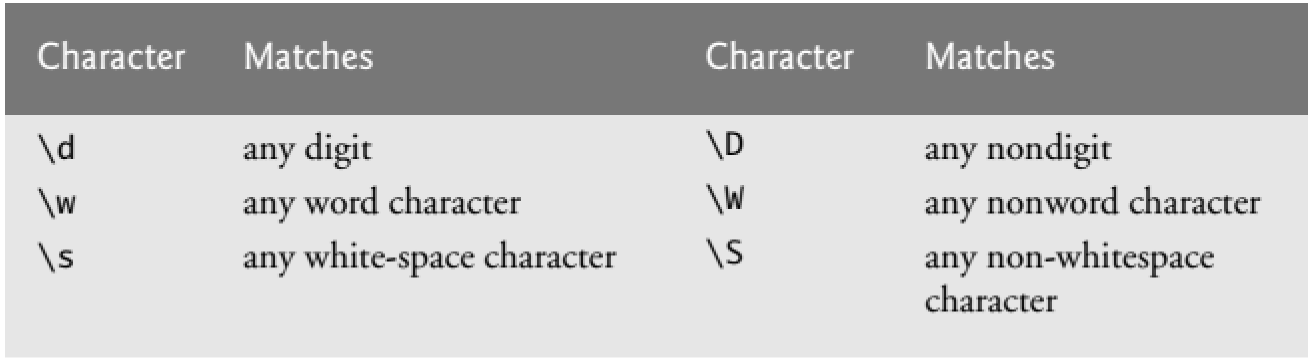


```
Enter a sentence and press Enter
This is a sentence with seven tokens
Number of elements: 7
The tokens are:
This
is
a
sentence
with
seven
tokens
```

- **Regular Expressions**

Regular Expressions

- Regular expression
  - Sequence of characters and symbols
  - Define set of strings
- Class `Pattern`
  - An immutable regular expression
- Class `Match`
  - A regular expression matching operation



Character	Matches	Character	Matches
`\d`	any digit	`\D`	any nondigit
`\w`	any word character	`\W`	any nonword character
`\s`	any white-space character	`\S`	any non-whitespace character

Fig. 14.19 | Predefined character classes.

Regular Expressions

- Example of Java Regular Expressions

```
import java.util.regex.*;
public class RegexExample1{
        public static void main(String args[]){
                //1st way
                Pattern p = Pattern.compile(".s");//. represents single character
                Matcher m = p.matcher("as");
                boolean b = m.matches();

                //2nd way
                boolean b2=Pattern.compile(".s").matcher("as").matches();

                //3rd way
                boolean b3 = Pattern.matches(".s", "as");

                System.out.println(b+" "+b2+" "+b3);
        }
}
```

Output

```
true true true
```

Reference: https://www.javatpoint.com/java-regex

Regular Expressions

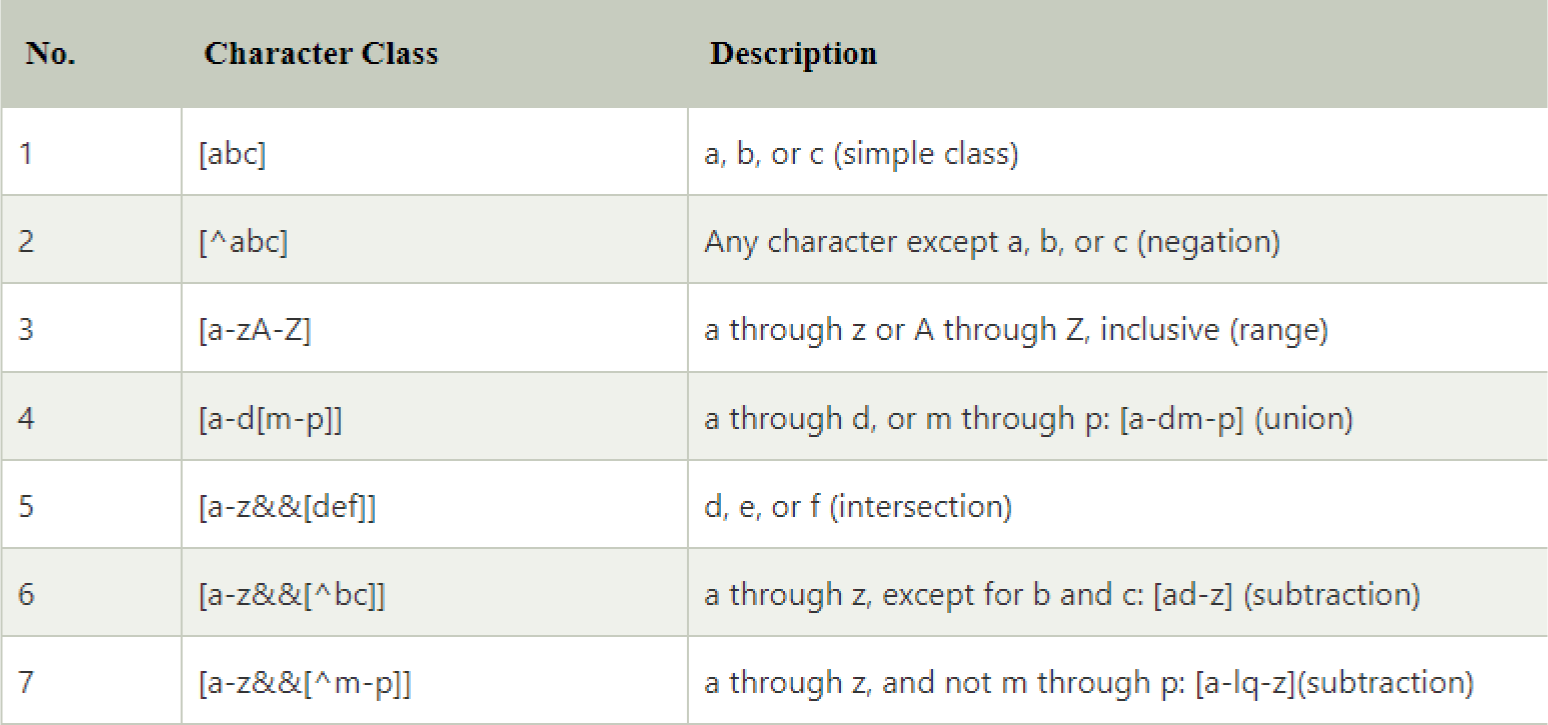
- Example 2:

```
import java.util.regex.*;
class RegexExample2{
	public static void main(String args[]){
		System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)
		System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)
		System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)
		System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)
		System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)
	}
}
```

Reference: https://www.javatpoint.com/java-regex

Regular Expressions

- Regex Character classes



No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

Reference: https://www.javatpoint.com/java-regex

Regular Expressions

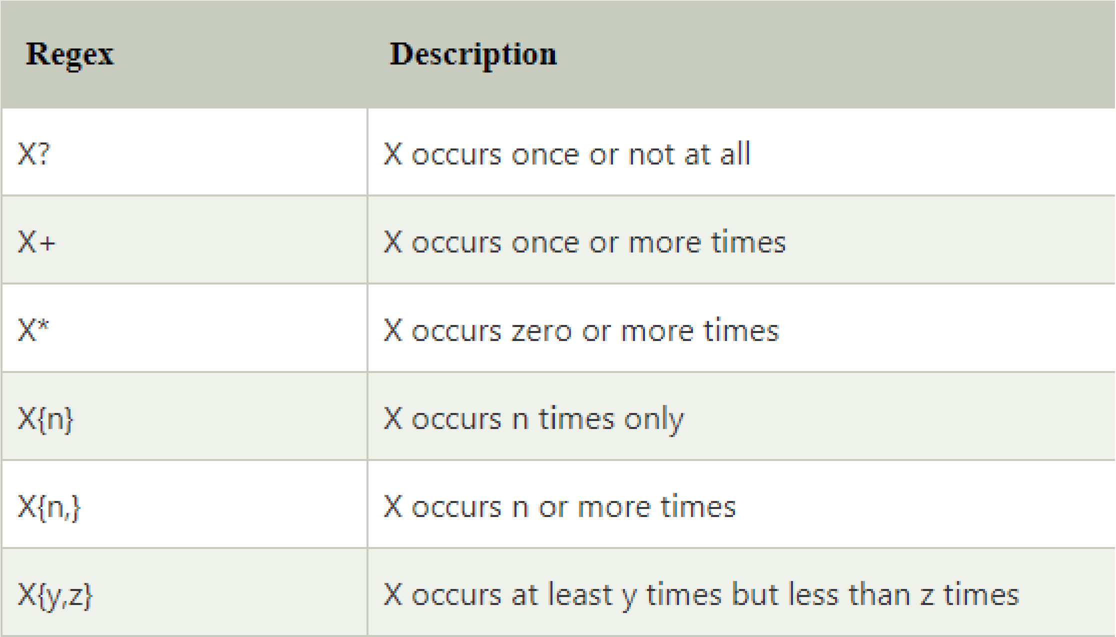
- Regex Character classes Example:

```
import java.util.regex.*;
class RegexExample3{
        public static void main(String args[]){
                System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
                System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)
                System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a comes more than once)
        }
}
```

Reference: https://www.javatpoint.com/java-regex

Regular Expressions

- Regex Quantifiers



Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

Reference: https://www.javatpoint.com/java-regex

Regular Expressions

- Example

```
import java.util.regex.*;
class RegexExample6{
        public static void main(String args[]){
                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true
                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)
                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true
                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)
        }
}
```

Reference: https://www.javatpoint.com/java-regex

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You

الجامعة السعودية الإلكترونية
SAUDI ELECTRONIC UNIVERSITY
2011-1432

College of Computing and Informatics
Data Science Pre-Master Program
CS230
Object Oriented Programming

CS230
Object Oriented Programming

Week 14

Recursion

Contents

1. Recursion Concepts
2. Using Recursion: Factorials
3. Using Recursion: Fibonacci Series
4. Recursion vs. Iteration
5. Towers of Hanoi

Weekly Learning Outcomes

- Demonstrate the concept of Recursion
- Differentiate between recursion and iteration in Java

Required Reading

1. Chapter 18 - (Java: How to program: Late Objects)

Recommended Reading

1. Chapter 7 - (Java The Complete Reference, Eleventh Edition)
2. Java Recursion:

https://www.programiz.com/java-programming/recursion

- Recursion Concepts

Recursion Concepts

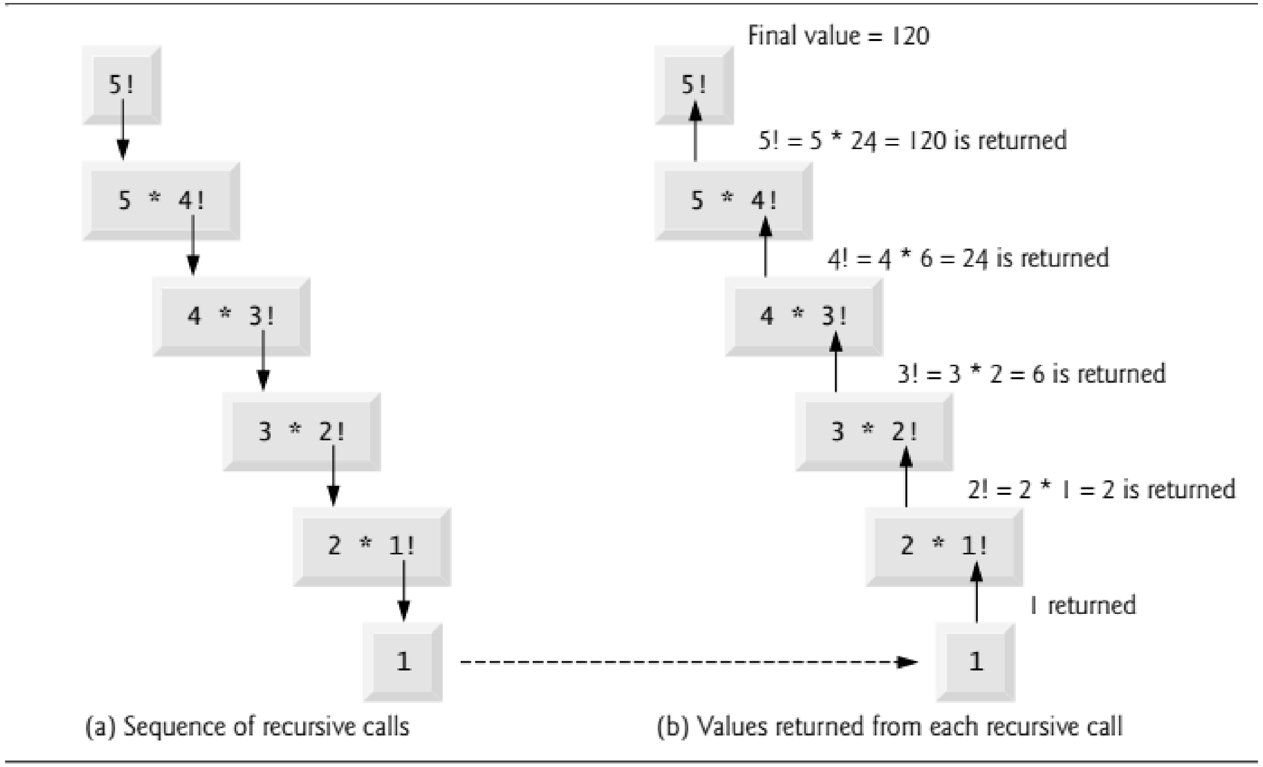
- Recursive method
  - Calls itself (directly or indirectly) through another method
  - Method knows how to solve only a *base case*
  - Method divides problem
    - Base case
    - Simpler problem
      - Method now divides simpler problem until solvable
  - Recursive call
  - Recursive step

- Using Recursion: Factorials

Using Recursion: Factorials

- the *factorial* of a positive integer n, written n!

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

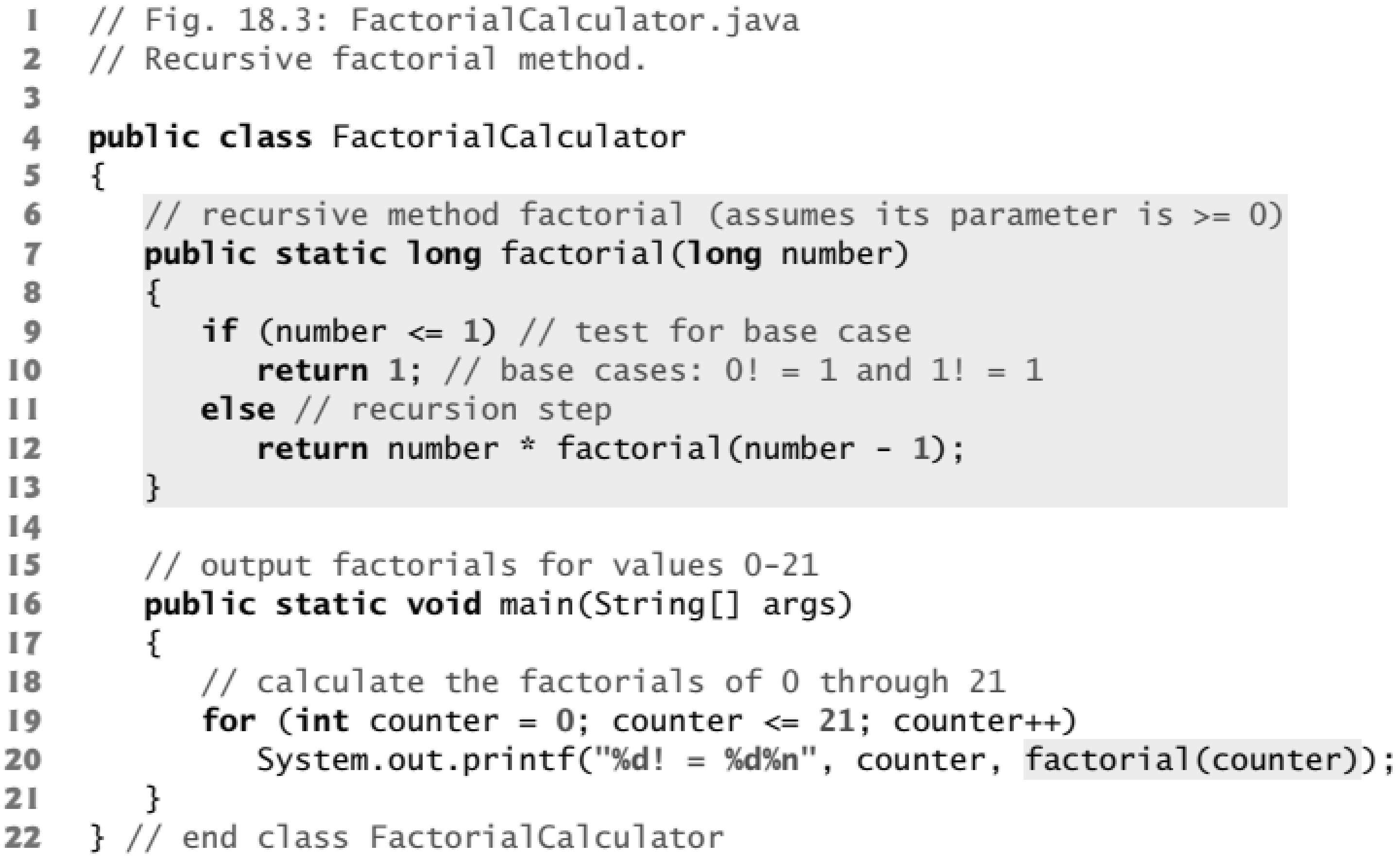
(a) Sequence of recursive calls

(b) Values returned from each recursive call

Fig. 18.2 | Recursive evaluation of 5!.

Using Recursion: Factorials

- Example:



```java
// Fig. 18.3: FactorialCalculator.java
// Recursive factorial method.

public class FactorialCalculator
{
   // recursive method factorial (assumes its parameter is >= 0)
   public static long factorial(long number)
   {
      if (number <= 1) // test for base case
         return 1; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number * factorial(number - 1);
   }

   // output factorials for values 0-21
   public static void main(String[] args)
   {
      // calculate the factorials of 0 through 21
      for (int counter = 0; counter <= 21; counter++)
         System.out.printf("%d! = %d%n", counter, factorial(counter));
   }
} // end class FactorialCalculator
```

Output

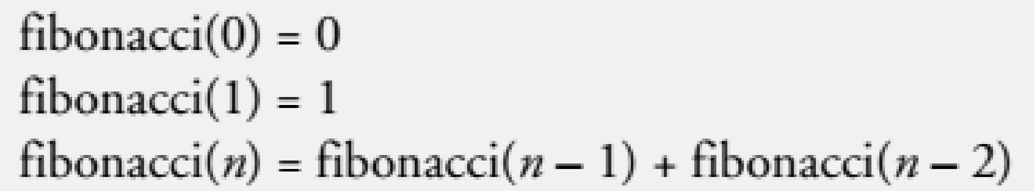
```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

- Using Recursion: Fibonacci Series

Using Recursion: Fibonacci Series

- The Fibonacci series,
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
  - It has the property that each subsequent Fibonacci number is the sum of the previous two.

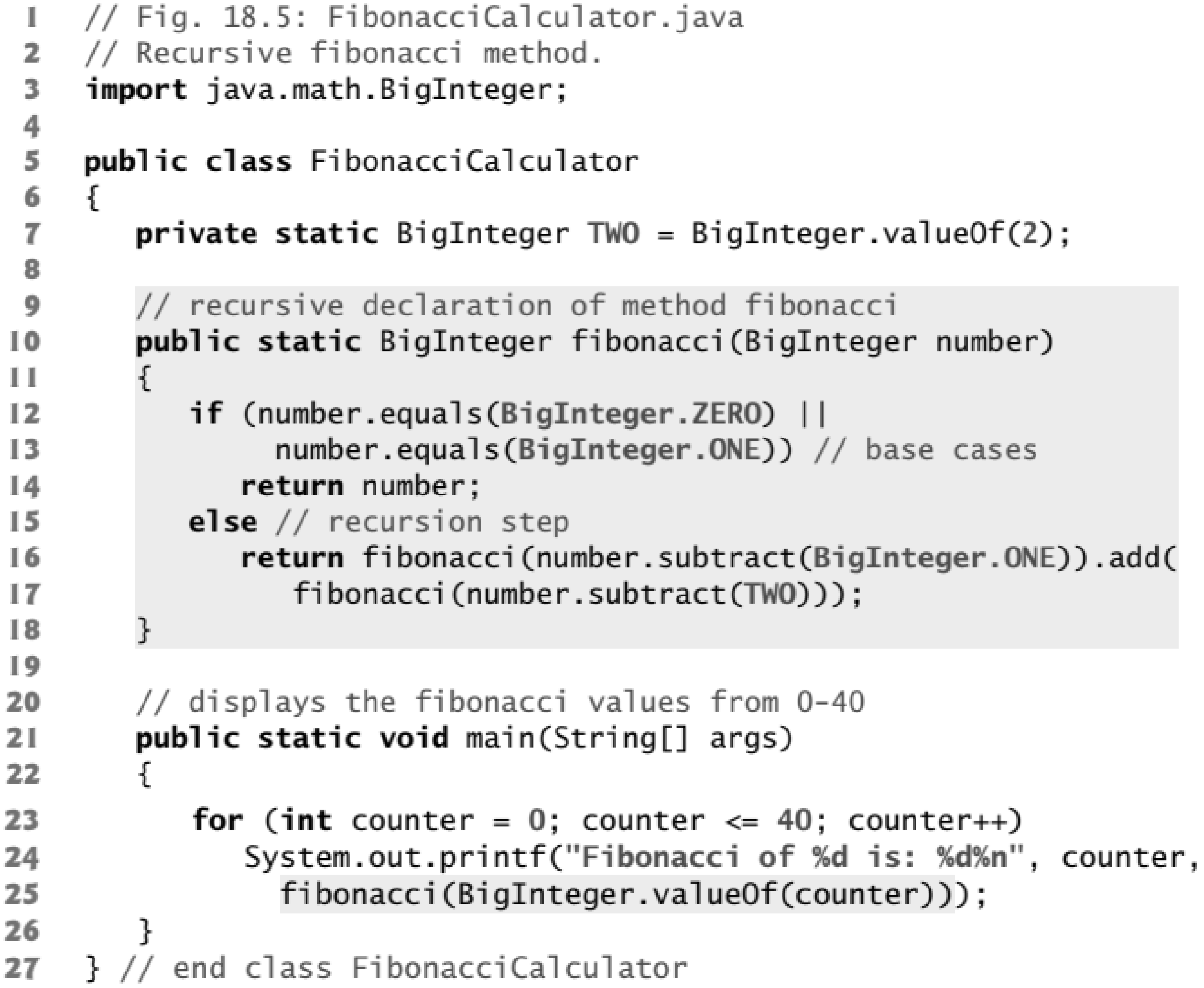
  - The Fibonacci series may be defined recursively as follows:



$$\text{fibonacci}(0) = 0$$
$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

Using Recursion: Fibonacci Series

- Example:



```java
// Fig. 18.5: FibonacciCalculator.java
// Recursive fibonacci method.
import java.math.BigInteger;

public class FibonacciCalculator
{
   private static BigInteger TWO = BigInteger.valueOf(2);

   // recursive declaration of method fibonacci
   public static BigInteger fibonacci(BigInteger number)
   {
      if (number.equals(BigInteger.ZERO) ||
            number.equals(BigInteger.ONE)) // base cases
         return number;
      else // recursion step
         return fibonacci(number.subtract(BigInteger.ONE)).add(
            fibonacci(number.subtract(TWO)));
   }

   // displays the fibonacci values from 0-40
   public static void main(String[] args)
   {
      for (int counter = 0; counter <= 40; counter++)
         System.out.printf("Fibonacci of %d is: %d%n", counter,
            fibonacci(BigInteger.valueOf(counter)));
   }
} // end class FibonacciCalculator
```

Output

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
...
Fibonacci of 37 is: 24157817
Fibonacci of 38 is: 39088169
Fibonacci of 39 is: 63245986
Fibonacci of 40 is: 102334155
```

Using Recursion: Fibonacci Series

- Example:

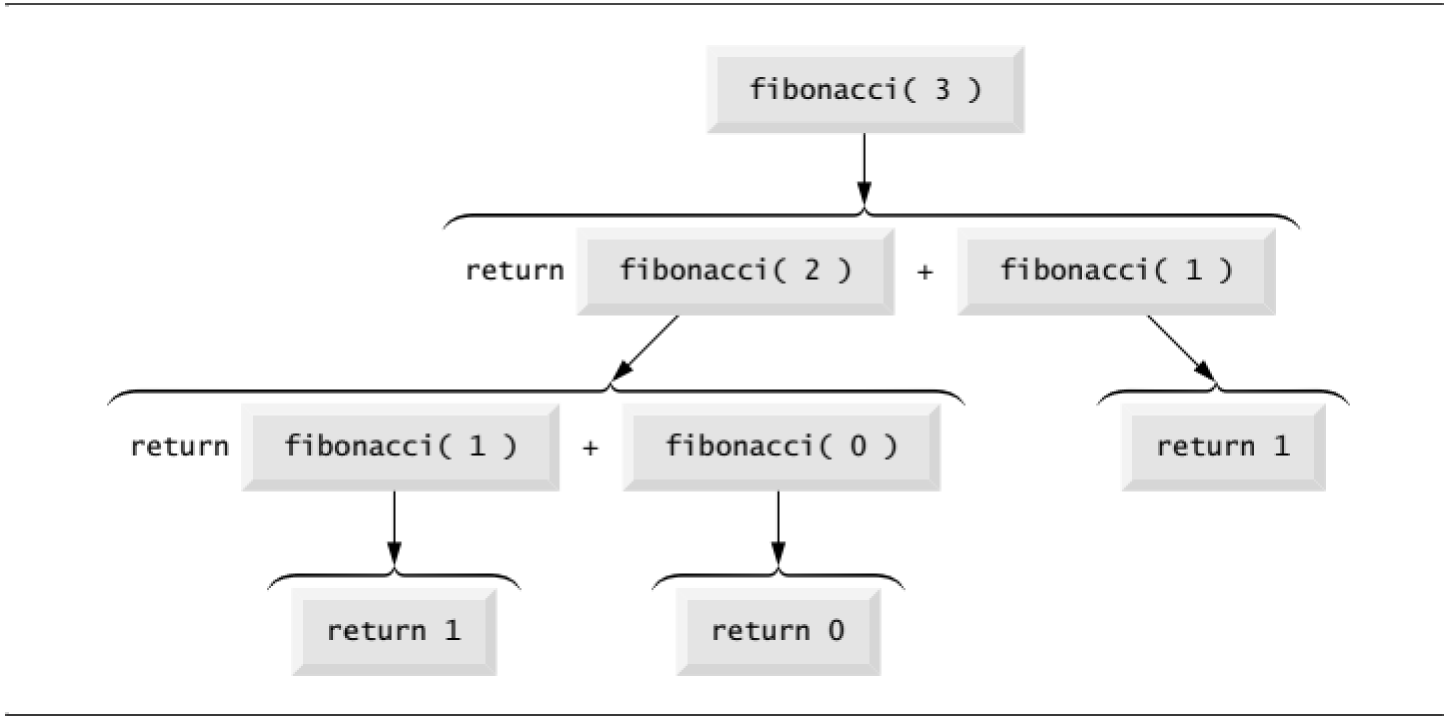

fibonacci(3)
return fibonacci(2) + fibonacci(1)
return fibonacci(1) + fibonacci(0)
return 1
return 1
return 0

Fig. 18.6 | Set of recursive calls for `fibonacci(3)`.

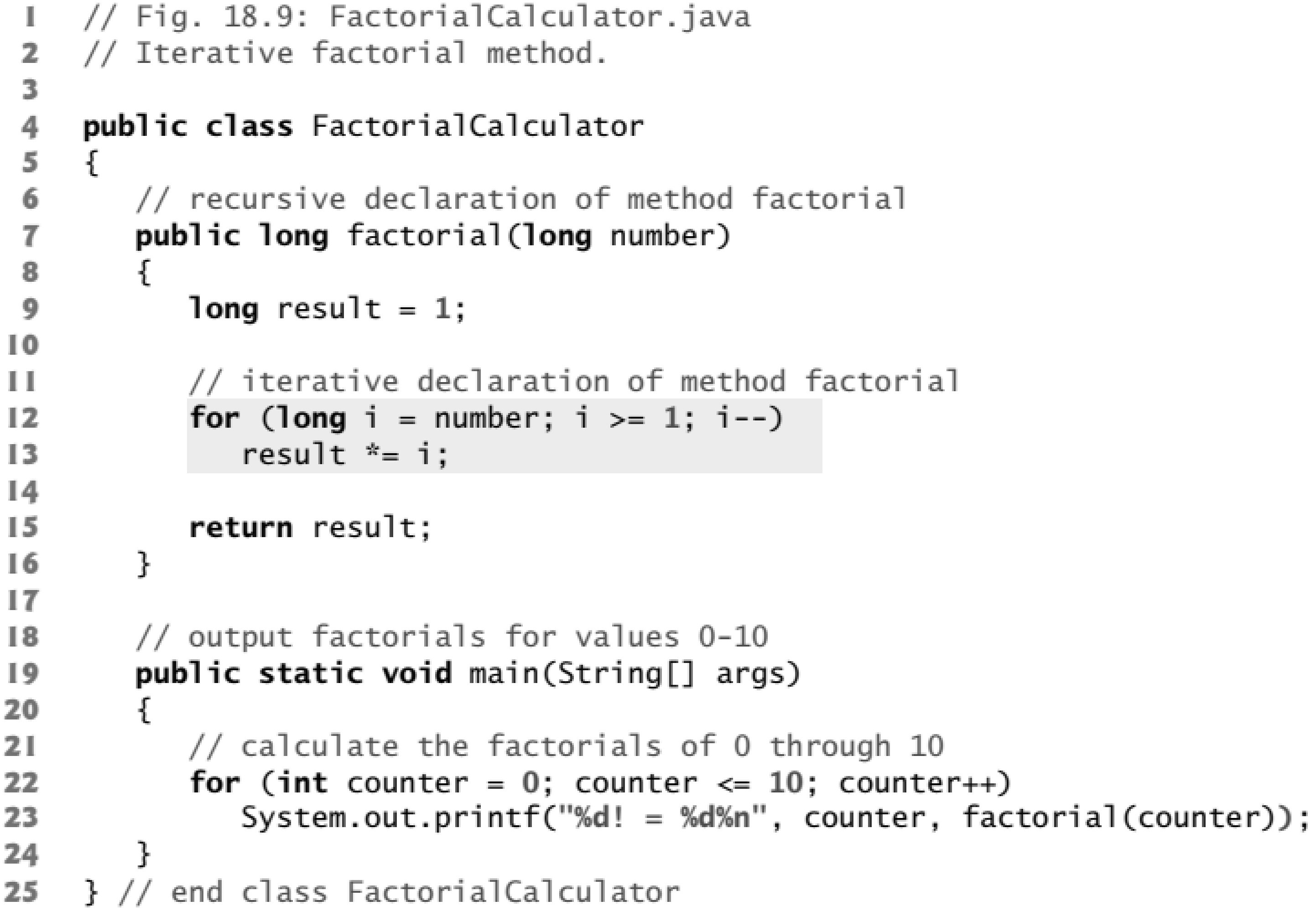
- Recursion vs. Iteration

Recursion vs. Iteration

Iteration	Recursion
Uses repetition structures (for, while or do…while)	Uses selection structures (if, if…else or switch)
Repetition through explicitly use of repetition structure	Repetition through repeated method calls
Terminates when loop-continuation condition fails	Terminates when base case is satisfied
Controls repetition by using a counter	Controls repetition by dividing problem into simpler one
	More overhead than iteration
	More memory intensive than iteration
	Can also be solved iteratively
	Often can be implemented with only a few lines of code

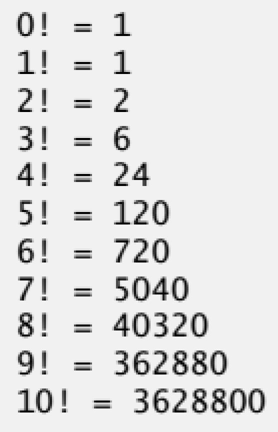
Recursion vs. Iteration

- Example: calculating Factorial using iteration



```java
// Fig. 18.9: FactorialCalculator.java
// Iterative factorial method.

public class FactorialCalculator
{
   // recursive declaration of method factorial
   public long factorial(long number)
   {
      long result = 1;

      // iterative declaration of method factorial
      for (long i = number; i >= 1; i--)
         result *= i;

      return result;
   }

   // output factorials for values 0-10
   public static void main(String[] args)
   {
      // calculate the factorials of 0 through 10
      for (int counter = 0; counter <= 10; counter++)
         System.out.printf("%d! = %d%n", counter, factorial(counter));
   }
} // end class FactorialCalculator
```

Output



```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

- Towers of Hanoi

Towers of Hanoi

- The Towers of Hanoi is one of the classic problems every budding computer scientist must grapple with.
    - It is an example of a problem that can be solved easily with Recursion than iteration

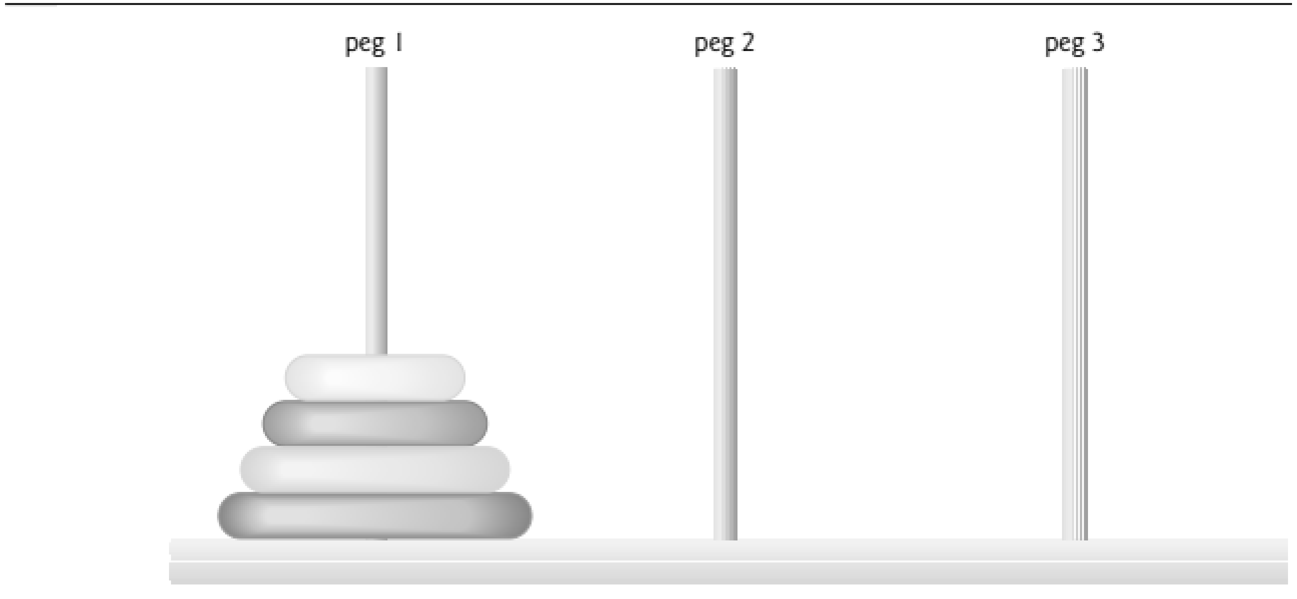


Fig. 18.10 | Towers of Hanoi for the case with four disks.

Towers of Hanoi

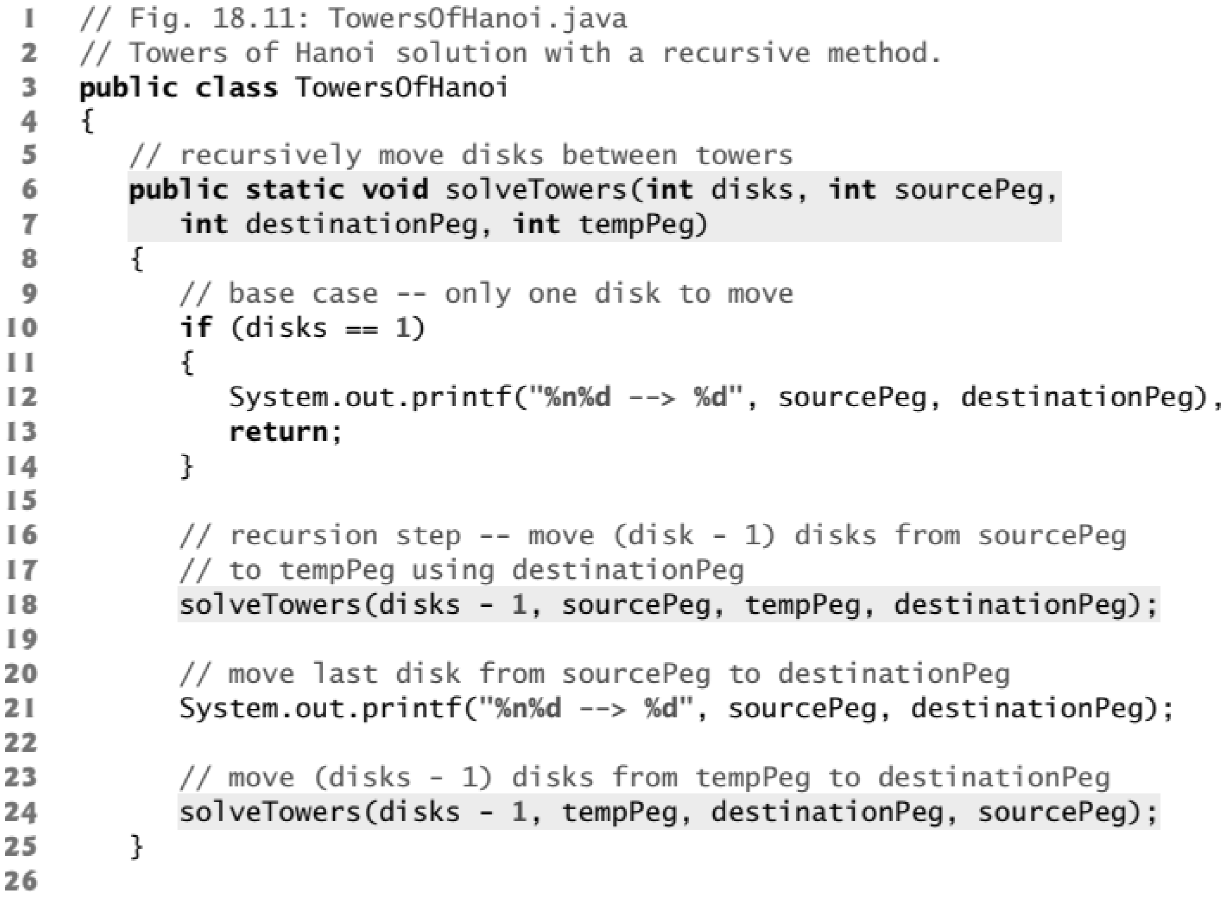
- Let's assume that the priests are attempting to move the disks from peg 1 to peg 3.
- Task:
  - Develop an algorithm that prints the precise sequence of peg to-peg disk transfers.

  - Moving *n* disks can be viewed in terms of moving only *n* – 1 disks

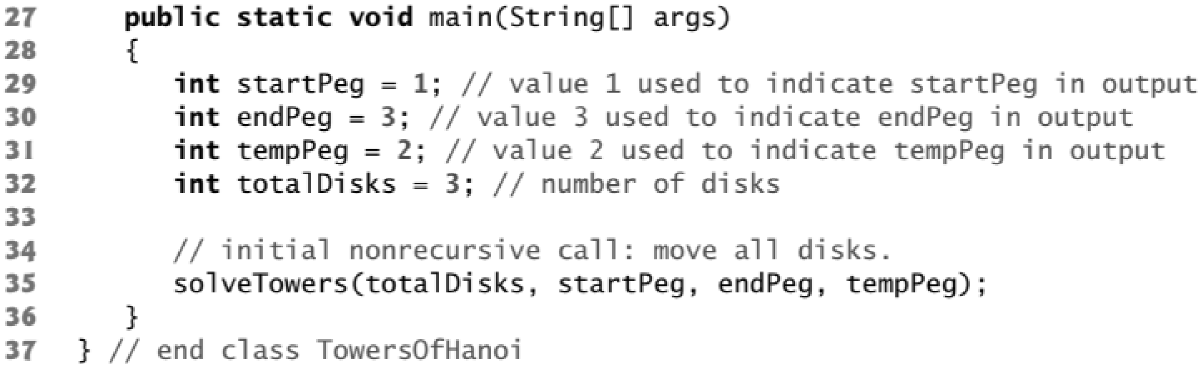
    1. Move *n* – 1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
    2. Move the last disk (the largest) from peg 1 to peg 3.
    3. Move *n* – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

Towers of Hanoi

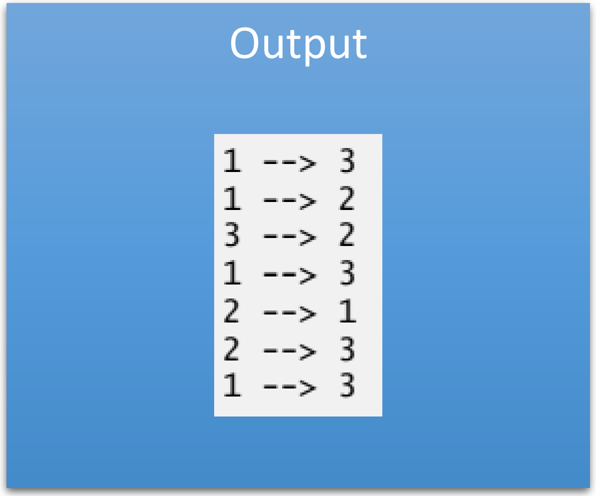
- Algorithm Implementation:



```
// Fig. 18.11: TowersOfHanoi.java
// Towers of Hanoi solution with a recursive method.
public class TowersOfHanoi
{
   // recursively move disks between towers
   public static void solveTowers(int disks, int sourcePeg,
      int destinationPeg, int tempPeg)
   {
      // base case -- only one disk to move
      if (disks == 1)
      {
         System.out.printf("%n%d --> %d", sourcePeg, destinationPeg),
         return;
      }

      // recursion step -- move (disk - 1) disks from sourcePeg
      // to tempPeg using destinationPeg
      solveTowers(disks - 1, sourcePeg, tempPeg, destinationPeg);

      // move last disk from sourcePeg to destinationPeg
      System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);

      // move (disks - 1) disks from tempPeg to destinationPeg
      solveTowers(disks - 1, tempPeg, destinationPeg, sourcePeg);
   }

```



```
   public static void main(String[] args)
   {
      int startPeg = 1; // value 1 used to indicate startPeg in output
      int endPeg = 3; // value 3 used to indicate endPeg in output
      int tempPeg = 2; // value 2 used to indicate tempPeg in output
      int totalDisks = 3; // number of disks

      // initial nonrecursive call: move all disks.
      solveTowers(totalDisks, startPeg, endPeg, tempPeg);
   }
} // end class TowersOfHanoi
```



Output

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

References

- Deitel, P. J., & Deitel, H. M. (2017). *Java: how to program late objects*. Pearson.

Thank You